

普通高等教育"十一五"国家级规划教材



21世纪大学本科计算机专业系列教材

王挺 周会平 贾丽丽 徐锡山 编著

C++程序设计（第3版）

http://www.tup.com.cn

- 根据教育部"高等学校计算机科学与技术专业规范"组织编写
- 与美国 ACM 和 IEEE CS *Computing Curricula* 最新进展同步

清华大学出版社

普通高等教育“十一五”国家级规划教材

21 世纪大学本科计算机专业系列教材

C ++ 程序设计(第 3 版)

王　挺　周会平　贾丽丽　徐锡山　编著

清 华 大 学 出 版 社
北　京

内 容 简 介

本书参照 ACM 和 IEEE CS CC2013 以及教育部高等学校计算机类专业教学指导委员会编制的《高等学校计算机科学与技术专业发展战略研究报告暨专业规范(试行)》的要求，面向大学计算机专业本科教学的需要，系统地介绍 C++ 程序设计语言的语法、语义和语用，使读者掌握结构化程序设计方法和面向对象程序设计方法。全书分为程序设计基础和面向对象程序设计两大部分。第一部分主要介绍类型、运算符与表达式、输入与输出、控制结构、函数、数组、指针、结构、联合、枚举、链表等程序设计的基本概念和结构化程序设计方法；第二部分主要介绍类与对象、类和对象的使用、操作符重载、继承、多态性、文件和流、异常和模板等面向对象程序设计的基本概念和程序设计方法。

本书注重知识的系统性和连贯性，在内容上注意与后续课程的衔接，强调严密的逻辑思维，突出程序设计方法的教学。

本书适合作为高等学校"程序设计基础"或者"高级语言程序设计"课程的教材，也可供广大自学人员学习参考。

图书在版编目（CIP）数据

C++ 程序设计/王挺等编著. --3 版. --北京：清华大学出版社，2015
21 世纪大学本科计算机专业系列教材
ISBN 978-7-302-39348-1

Ⅰ. ①C…　Ⅱ. ①王…　Ⅲ. ①C 语言－程序设计－高等学校－教材　Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2015)第 024973 号

责任编辑：张瑞庆
封面设计：
责任校对：焦丽丽
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：28.25　　**字　　数**：688 千字
版　　次：2005 年 1 月第 1 版　2015 年 2 月第 3 版　　**印　　次**：2015 年 2 月第1 次印刷
印　　数：1～　000
定　　价：　.00 元

产品编号：058874-01

前 言

FOREWORD

本教材参照 ACM 和 IEEE CS CC2013 以及教育部高等学校计算机类专业教学指导委员会编制的《高等学校计算机科学与技术专业发展战略研究报告暨专业规范(试行)》(下面简称《规范》)的要求,力图通过教学,使学生掌握结构化程序设计方法和面向对象程序设计方法,掌握 C++ 程序设计语言的语法、语义和语用,能够熟练运用 C++ 语言解决一般问题。此外,通过课程学习,使学生掌握学习高级程序设计语言的一般方法,养成良好的程序设计风格,对软件工程有初步的认识。本教材虽然以讲授 C++ 为主,但并非单纯讲授一门语言,而是更加强调对程序设计方法的掌握和程序设计风格的养成,为学生今后继续学习其他高级程序设计课程打下牢固的基础。

本书的编写参照 CC2013 和《规范》的知识体系,覆盖或涉及其中程序设计基础(PF)、程序设计语言(PL)、算法和复杂性(AL)3 个领域(area)的多个知识单元(unit)。由于部分知识单元的内容将在其他课程(如数据结构、算法设计与分析、编译原理等)中详细介绍,因此本教材并未深入讲解。

本教材覆盖或涉及 CC2013 和《规范》的如下知识。

1) 程序设计基础(PF)

- 程序设计基本结构。
- 算法和问题求解。
- 基本数据结构。
- 递归。
- 事件驱动程序设计。

2) 程序设计语言(PL)

- 程序设计语言概论。
- 声明和类型。
- 抽象机制。
- 面向对象程序设计。

3) 算法和复杂性(AL)

- 算法策略。
- 基本算法。

本书参考了国外著名高校教材,结合国内高校教学的需要和学生的特点,力求形成如下特色。

(1) 注重知识的系统性和连贯性。本教材面向大学本科计算机专业的学生,参照 CC2013 和《规范》的要求,在教学内容上注意与后续课程(如数据结构、算法设计与分析、编译原理、软件工程等)的衔接。

(2) 突出程序设计方法,强调严密的逻辑思维。本教材并非单纯讲授 C++ 程序设计语言,而是在介绍 C++ 程序设计语言的基础上,系统地讲解程序设计方法,包括结构化程序设计方法和面向对象程序设计方法,特别是后者。程序设计方法是本书的讲授重点。因此,在教材的示例程序中,一般都配有分析、解释和经验总结等,帮助学生领会程序设计的方法和思想。

(3) 注重实践能力的培养。本教材提供丰富的典型例题,每章之后配有难易和综合程度各不相同的习题。教材中还适当地穿插介绍一些编程技巧和软件设计经验,使学生能够从实践的角度更好地学习和掌握 C++ 程序设计方法。

(4) 注重程序设计风格的养成。良好的程序设计风格是保证软件质量的基础。本教材除了有专门章节介绍程序设计风格外,全书始终突出对程序设计风格的要求,并介绍了从分析、设计到编程如何实现良好程序设计风格的方法。本教材的示例程序即按照这种方法实现,在变量命名、源程序格式等方面都贯彻统一的风格。

C++ 是当今最流行的一种高效实用的高级程序设计语言,应用十分广泛。它也是一门复杂的语言,与 C 语言兼容,既支持结构化的程序设计方法,也支持面向对象的程序设计方法,因而成为编程人员最广泛使用的工具。在学习 C++ 的基础上,可以进一步学习其他程序设计语言,C++ 架起了通向强大、易用、真正的应用软件开发的桥梁。本书共分为两大部分:第一部分,第 1 章至第 9 章是基础部分,主要介绍 C++ 程序设计语言、程序结构和结构化程序设计基础;第二部分,第 10 章至第 18 章是面向对象程序设计部分,它建立在 C++ 程序设计基础之上,讲述了面向对象程序设计方法。书中带 * 的章节为选讲内容,可以根据实际情况取舍。

本书是作者根据多年教学实践的经验编写而成,适合作为大学本科计算机专业和非计算机专业的“程序设计基础”或者“高级语言程序设计”课程的教材,也可供广大读者自学参考。由于作者水平有限,书中可能存在缺点和错误,恳请广大读者批评指正。

作者
国防科技大学计算机学院
2015 年 1 月

目　录

CONTENTS

第 1 章 计算机与程序设计

【学习内容】

本章介绍计算机与程序的基本概念。主要内容包括：

◆ 计算机系统的基本组成和工作原理。

◆ 程序设计的基本过程和方法。

◆ 程序设计语言的基本概念及其发展历史。

◆ 结构化程序设计方法和面向对象程序设计方法概述。

◆ C 语言与 C++ 语言。

◆ 典型的 C++ 编程环境、开发过程和简单的 C++ 程序结构。

◆ 程序设计风格。

【学习目标】

◆ 了解计算机系统、程序设计语言、程序设计的基本概念。

◆ 通过简单示例掌握 C++ 程序的结构。

◆ 理解程序设计风格的意义。

1.1 计算机系统概述

1.1.1 什么是计算机系统

根据中国大百科全书的定义，计算机系统（computer system）是按人的要求接收和存储信息，自动进行数据处理和计算，并输出结果信息的机器系统。计算机是脑力的延伸和扩充，是近代科学的重大成就之一。计算机系统由硬件系统和软件系统组成，前者是借助电、磁、光、机械等原理构成的各种物理部件的有机组合，是系统赖以工作的实体，如 CPU、显示器、内存、硬盘和键盘等；后者是各种程序和文档，用于指挥全系统按指定的要求进行工作，程序是对计算任务的处理对象和处理规则的描述，而文档是与软件研制、维护和使用有关的资料。随着计算机技术的迅速发展，硬件的成本在不断地下降，个人计算机得到了广泛应用和普及。这一趋势也为计算机在信息处理方面提出了更高的要求——设计实现更方便、功能更加强大的应用程序。但是，相对于硬件技术的发展，软件的开发技术明显滞后。现阶段的软件设计更多地依赖于人，而软件设计的方法，包括程序设计的方法，是提高软件生产率和质量的关键。

1.1.2 计算机硬件

计算机硬件系统是计算机系统快速、可靠、自动工作的基础。计算机硬件就其逻辑功能来说,主要是完成信息变换、信息存储和信息传送等处理功能,它为计算机软件提供具体实现的基础。计算机硬件系统主要由运算器、主存储器、控制器、输入设备、输出设备和辅助存储器等功能部件组成。

1. 运算器

运算器的主要功能是对数据进行算术运算和逻辑运算。整个运算过程是在控制器的控制下自动进行的。操作时,运算器从主存储器取得运算数据,经过指令指定的运算处理,所得运算结果或者留在运算器内以备下次运算时使用,或者写入主存储器。

2. 主存储器

主存储器的主要功能是存储二进制信息。它与运算器、控制器等快速部件直接交换信息。从主存储器中应能快速读出信息,并送到其他功能部件中去,或者将其他功能部件处理过的信息快速写入主存储器。

3. 控制器

控制器的功能主要是按照机器代码程序的要求,控制计算机各功能部件协调一致地动作,即从主存储器取出程序中的指令,并对该指令进行分析和解释,然后向其他功能部件发出执行该指令所需要的各种时序控制信号,再从主存储器取出下一条指令执行,如此连续运行下去,直到程序执行完为止。计算机自动工作的过程就是逐条执行程序中指令的过程。控制器与运算器一起构成中央处理器,中央处理器与主存储器一起构成处理机。

4. 输入设备

输入设备主要是将用户信息(数据、程序等)变换为计算机能识别和处理的信息形式。输入设备种类很多,如纸带阅读机、软磁盘机、汉字输入设备、键盘输入设备等。其工作特点是将人工编制的程序和原始数据,在某种媒介物上以二进制编码形式表现,如纸带上穿孔和不穿孔分别表示1和0,磁表面上磁化方向不同以区别1和0等。载有信息的媒介物通过相应的输入设备,将信息变换成电信号为计算机接收,并存入存储器。

5. 输出设备

输出设备主要是将计算机中二进制信息变换为用户所需要并能识别的信息形式。输出设备种类很多,如打印机、凿孔输出机、汉字输出设备、绘图仪、显示终端、声音输出设备等。其工作特点与输入设备正好相反,是将计算机中二进制信息经过相应变换,成为用户需要的信息形式,记录在媒介物上或通过各种媒体形式表现出来供用户使用。输出的信息形式多为十进制数字、字符、图形、表格、声音等。

6. 辅助存储器

辅助存储器主要是存放主存储器难以容纳、又为程序执行所需要的大量文件信息,其特点是存储容量大,存储成本低,但存取速度较慢。它不能直接与中央处理器交换信息。辅助存储器一般为磁盘、磁带、光盘和优盘等。

1.1.3 计算机软件

光有硬件计算机还不能工作,要使计算机能解决各种实际问题,还必须有软件的支持。

软件是用户与硬件之间的接口界面。使用计算机就必须针对待解的问题拟定算法,用计算机所能识别的语言对有关的数据和算法进行描述,即必须编写程序。用户通过软件与计算机进行交互。软件规定了计算机系统如何进行工作,包括各项计算任务内部的工作内容和工作流程,以及各项任务之间的调度和协调。软件是计算机系统结构设计的重要依据。为了方便用户,在设计计算机系统时,必须通盘考虑软件与硬件的结合,以及用户的要求和软件的要求。

计算机软件通常分为系统软件、支撑软件和应用软件。

1. 系统软件

系统软件居于计算机系统中最靠近硬件的一层。其他软件一般都通过系统软件发挥作用。系统软件与具体的应用领域无关,如操作系统和编译程序等。操作系统(operating system)负责管理系统的各种资源、控制程序的执行。编译程序(compiler)把程序员用高级语言书写的程序翻译成与之等价的、可执行的低级语言程序,如机器语言程序。在任何计算机系统的设计中,系统软件都要优先予以考虑。

2. 支撑软件

支撑软件是用于支撑其他软件开发和维护的软件。随着计算机科学技术的发展,软件的开发和维护代价在整个计算机系统中所占的比重很大,远远超过硬件。因此,支撑软件的研究具有重要意义,可直接促进软件的发展。当然,编译程序、操作系统等系统软件也可算做支撑软件。但是,20 世纪 70 年代中后期发展起来的软件支撑环境可看成是现代支撑软件的代表,主要包括环境数据库、各种接口软件和工具组。三者形成整体,协同支撑其他软件的开发。

3. 应用软件

应用软件是特定应用领域专用的软件,如天气预报用的计算软件就是一种应用软件。对于具体的应用领域,应用软件的质量往往成为影响实际效果的决定性因素。随着计算机技术的发展,特别是互联网出现以来,计算机在整个社会中应用的深度和广度都在迅速提高,各种应用软件的复杂程度也越来越高,不断地推动软件技术的发展。

上述分类不是绝对的,而是互相交叉和变化的。有些软件,如操作系统和编译程序,既可看做是系统软件,又可看做是支撑软件。它们在一个系统中是系统软件,而在另一个系统中却是支撑软件;也可以在同一系统中既是系统软件,又是支撑软件。系统软件和应用软件之间也有类似情况。有的软件,如数据库管理系统、网络软件和图形软件,原来作为应用软件,后来又作为支撑软件。而且系统软件、支撑软件和应用软件三者的开发技术基本相同。因此,这三者既有分工,又有结合,并不截然分开。

1.2 程序设计基本概念

1.2.1 问题求解过程

在计算机中,一切信息处理都要受程序的控制,数值数据如此,非数值数据也是如此。因此任何问题求解(problem solving)最终要通过执行程序来完成。根据传统的(面向过程)程序设计方法,把计算机的应用需求转变为可在计算机上运行的程序,一般要经历问题定义、算法设计、程序编码、程序测试等步骤。

1. 问题定义

问题定义(problem definition)的目的就是明确拟解决的问题,写出求解问题的规格说明(specification)。主要内容包括用户要求的输入输出的数据及其形式、求解问题的数学模型或者对数据处理的需求、程序的运行环境等。在软件的开发过程中,需求分析完成的就是问题定义,即明确拟开发的软件的功能需求。

2. 算法设计

算法设计(algorithm design)是指把问题的数学模型或处理需求转化为计算机的解题步骤。算法设计的好坏直接影响着程序的质量。对于大型的软件开发,设计是一个非常复杂而重要的阶段,通常还要进一步分为概要设计和详细设计两个阶段。在概要设计阶段,主要是根据软件需求规格说明,建立目标软件系统的总体结构,设计全局数据结构,规定设计约束,制定组装测试计划等;而在详细设计阶段,主要是逐步细化概要设计所生成的各个模块,并详细描述程序模块的内部细节(数据结构、算法、工作流程等),形成可编程的程序模块。

3. 程序编码

程序编码(program coding)的主要任务是用选定的某种程序设计语言将前一步设计出来的算法实现为能在计算机上运行的程序。在软件开发过程中,编码的工作是严格根据详细设计规格说明而进行的,所以软件的设计应当尽可能做到详细、正确和完整。

4. 测试和调试

测试和调试(testing and debugging)的主要目的在于发现(通过测试)和纠正(通过调试)程序中的错误。只有经测试合格的程序才能交付用户使用。在软件开发过程中,通过对编写的程序进行调试和测试,以验证程序与详细设计文档的一致性,从而确保程序实现需求规格说明规定的功能,即解决了所定义的问题。

1.2.2 算法与程序

1. 算法

算法是求解问题类的、机械的、统一的方法,它由有限个步骤组成,对于问题类中的每个给定的具体问题,机械地执行这些步骤就可以得到问题的解答。算法的这种特性,使得计算不仅可以由人,而且可以由计算机来完成。用计算机解决问题的过程可以分成3个阶段:分析问题、设计算法和实现算法。要让计算机解决问题之前,必须先对问题进行分析,提出解决问题的办法,然后建立此问题的计算步骤,最后在计算机上实现。

算法具有下列特征:

- 有穷性。一个算法在执行有穷个计算步骤后必须终止。
- 确定性。一个算法给出的每一个计算步骤必须是精确定义、无二义性的。
- 能行性。算法中要执行的每一个计算步骤都可以在有限时间内做完。
- 输入。要求输入一个或多个输入信息;有的算法可能不要求输入,这时输入数据取自某一特定集合。
- 输出。一个算法一般有一个或多个输出信息。

算法与计算机没有必然的关系,可以用多种方法来描述算法。主要有以下3种。

1) 文字描述

文字描述即用自然语言(汉语、英语等)来描述算法。采取这种描述方法,可以使得算法

易读易理解。例如，下面是求解两个整数的整商的算法的文字描述。

① 读入两个整数，即被除数和除数。

② 如果除数等于 0，则输出除数为 0 的错误信息。

③ 否则，计算被除数和除数的整商，并输出计算结果。

2）图形描述

也可以采取图形描述的方法来描述算法，主要包括流程图（又称框图）、盒图（又称 N-S 图）、问题分析图（又称 PAD 图）等。这里主要介绍流程图。流程图是对算法逻辑顺序的图形描述。如用长方形表示计算公式，用菱形框表示条件判断等。此方法形象、清晰，画法简单，格式自由，不必考虑太多的机器细节或程序细节，其主要缺点是计算机很难直接识别。

流程图采用一些图框表示各种操作，形象直观，易于理解。ANSI（American National Standard Institute，美国国家标准化协会）规定了一些常用的流程图符号，已被普遍采用。主要的流程图符号如图 1-1 所示。

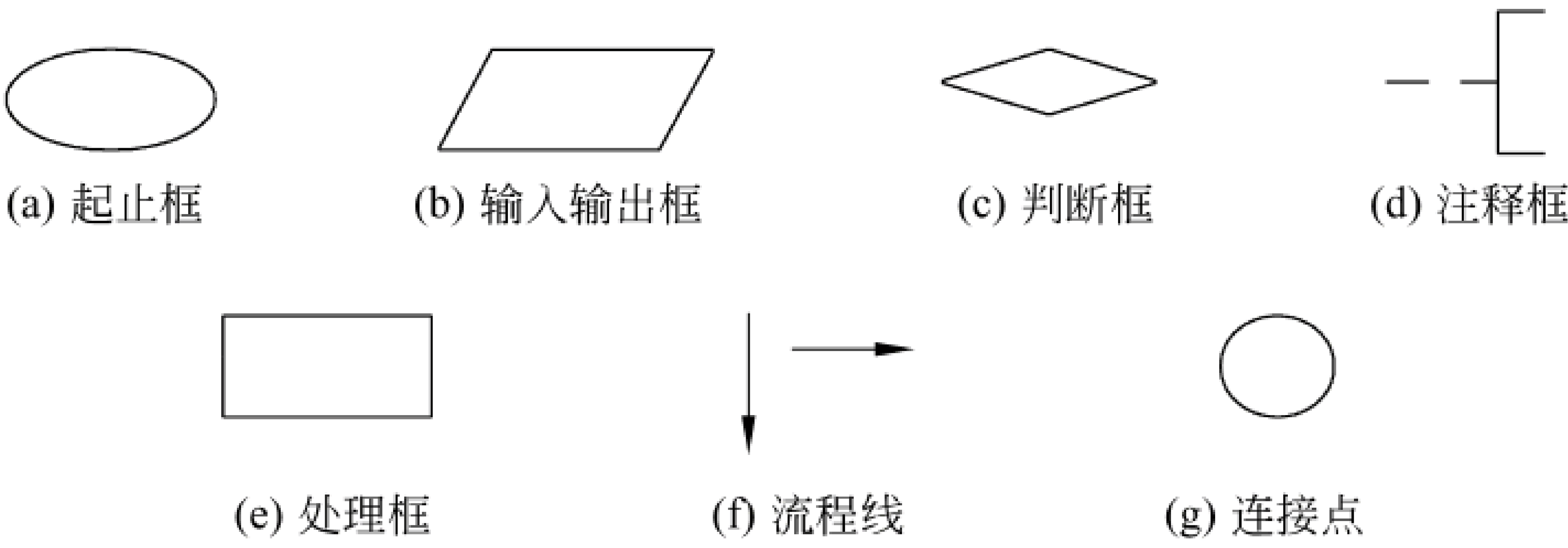


图 1-1　流程图符号

流程图是表示算法的较好工具。一个流程图包括：表示相应操作的框，带箭头的流程线，框内外必要的说明文字。

注意：流程线必须加箭头，以说明程序执行的先后次序，如果不画箭头就难以判定各框的执行顺序。

图 1-2 是求解两个整数的整商算法的流程图描述。

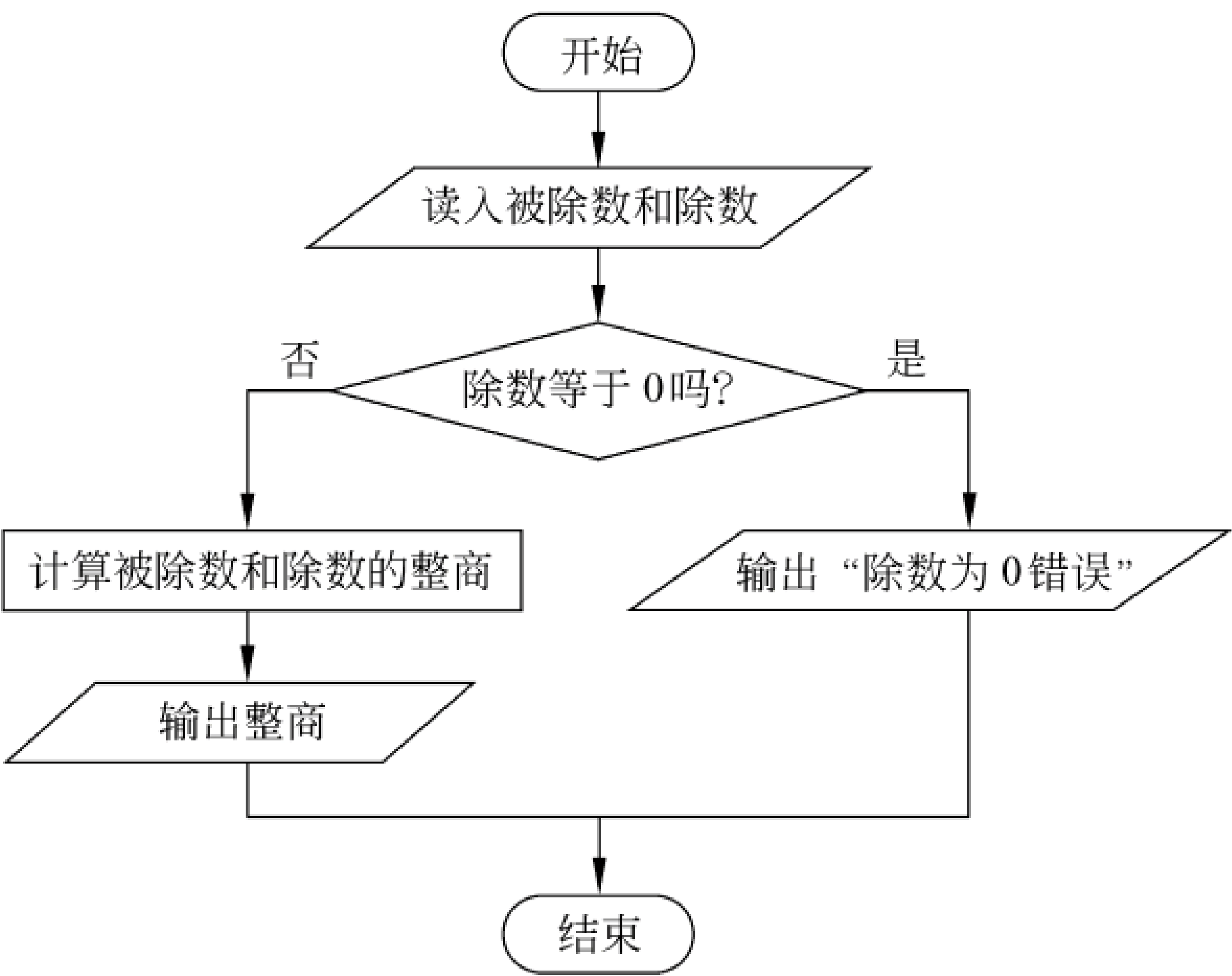


图 1-2　求解两个整数的整商算法的流程图

3) 程序设计语言描述

算法还可以用程序设计语言来描述,即将算法实现为计算机程序。

下面是用C++程序设计语言描述的求解两个整数的整商的算法。

【例1-1】 求两个整数的整商。

```
//ex1_1.cpp: 求两个整数的整商
#include<iostream.h>

int main()
{
    int dividend, divisor, quotient;                /* variables declaration */

    cout<<"Please enter the dividend:"<<endl;       /* prompt for dividend */
    cin>>dividend;                                  /* read the dividend */
    cout<<"Please enter the divisor:"<<endl;        /* prompt for divisor */
    cin>>divisor;                                   /* read the divisor */

    if (divisor==0)
        cout<<"Error: the divisor cannot be 0."<<endl;
                                                    /* output the error message */
    else
    {
        quotient=dividend/divisor;                  /* compute the quotient */
        cout<<"Quotient is "<<quotient<<endl;       /* output the result */
    }

    return 0;                                           /* indicate that program ended successfully */
}
```

2. 程序

程序是对计算任务的处理对象和处理规则的描述。所谓处理对象就是数据,包括数字、文字和图像等;而处理规则一般指处理动作和步骤。在低级语言中,程序是由一组指令和有关的数据组成。而在高级语言中,程序是由一组说明语句和执行语句组成。程序是程序设计中基本的概念,也是软件中基本的概念。程序的质量决定了软件的质量。

用自然语言描述的算法不能直接在计算机上执行,因为采取这种方式描述的算法,计算机并不理解。正像人们之间通过语言进行沟通一样,要让计算机完成工作,就必须使用计算机能够理解的语言,这种机器能理解的语言称为计算机语言。将算法用某种特定的计算机语言表达出来,输入到计算机,这便是程序设计。程序是用计算机语言描述的算法。

程序规定了计算机执行的动作和动作的顺序。程序应包括以下两方面的内容。

(1) 对数据的描述。在程序中要指定数据的类型和数据的组织形式,即数据结构。

(2) 对操作的描述。即操作步骤,说明如何对数据进行处理。

程序从本质上来说是描述一定数据的处理过程。著名的计算机科学家Niklaus Wirth提出了一个公式:

程序=数据结构+算法

3. 算法设计和程序编码

一些程序员,尤其是程序设计初学者,常常认为程序设计就是用某种程序设计语言编写

代码，这其实是错误的认识。上述工作应该被看成是程序编码，它是在程序的设计工作完成之后才开始的。以建筑设计为例，建筑设计这个过程不涉及砌砖垒瓦的具体工作，这些工作是在建筑施工阶段进行的。只有在完成了建筑设计，有了设计图纸之后，施工阶段才能开始。如果不进行设计，直接施工，很难保证房屋能按质按量建造完成。同样，在程序或软件的设计中，一定要先分析问题、设计解决问题的算法，然后再使用程序设计语言进行具体的编码。设计阶段主要完成求解问题的数据结构和算法的设计，设计完成的好坏直接影响着后面的编码质量。

1.2.3 程序设计语言

程序设计语言是用于书写计算机程序的语言。程序语言的基本功能是描述数据和对数据的运算。程序设计语言不同于汉语和英语等自然语言，它是人工语言。程序设计语言的定义由 3 个方面组成，即语法、语义和语用。语法表示程序的结构或形式，即表示构成语言的各个单位之间的组合规律，但不涉及这些单位的特定含义，也不涉及使用者。语义表示程序的含义，即表示按照各种方法所表示的各个单位的特定含义，但不涉及使用者。语用则表示程序与使用者的关系。语言的好坏不仅影响到其使用是否方便，而且涉及程序人员所写程序的质量。

程序员用各种程序设计语言编写指挥计算机进行工作的指令，有些指令是能够直接被计算机执行的，而有些指令则需要通过中间的翻译过程。当今使用的程序设计语言很多，可以分为低级语言和高级语言两大类，其中低级语言包括机器语言和汇编语言。

1. 机器语言

机器语言(machine language)是表示成数码形式的基本机器指令集。所有计算机只能直接执行本身的机器语言指令。机器语言程序通常由一组指令组成，每条指令指示计算机完成一个基本操作，这些指令以二进制数值字串形式表述。用机器语言编写程序复杂、繁琐和冗长。下面的机器语言程序把两个整数相加，并把结果保存在总和中。

```
0001  01  00  00001111
0011  01  00  00001100
0100  01  00  00010011
```

2. 汇编语言

随着计算机的普及，用机器语言编程对大多数程序员来说都是繁琐而痛苦的，为此设计了汇编语言，它是对机器语言进行符号化的结果。汇编语言使得程序员能够使用类似英语缩写的助记符来编写程序，从而摆脱了复杂、繁琐的二进制数据。下面的汇编源程序也是把两个整数相加，并把结果保存在总和中，但它比相应的机器语言程序清晰得多。

```
MOV a, R1
ADD b, R1
MOV R1, sum
```

用汇编语言编写程序比用机器语言更直观、更易于理解。但是汇编语言并不能被计算机直接执行，为此人们开发了相应的翻译程序——汇编程序(也称为汇编器)，它能把汇编语言编写的源程序转换为机器语言程序，从而可以在计算机上运行。

3. 高级语言

随着汇编语言的出现,计算机的应用范围迅速扩大,但是仍然难以克服低级语言所存在的问题,编写的程序还是难以理解,不便于维护,编写程序的效率非常低,开发复杂大型软件的难度非常大。为了提高编程效率,人们在汇编语言的基础上,开发出了高级程序设计语言,简称高级语言(high level language)。高级语言的表示方法要比低级语言更接近于待解问题的表示方法,其特点是在一定程度上与具体机器无关,易学、易用、易维护。如同汇编语言程序一样,使用高级语言编写的程序也不能直接在计算机上执行,需要通过编译程序把高级语言程序翻译成相应的机器语言程序,才能在计算机上运行(如果转换成汇编语言,还需借助汇编程序翻译成机器语言,才能运行)。一般来说,一个高级语言程序单位要对应多条机器指令,相应的编译程序所产生的目标程序往往功效较低。

同样是上面的问题,把两个整数相加,并把结果保存在总和中,用高级语言编写起来非常简单。

```
sum=a+b
```

显然,高级语言更接近于数学语言和自然语言,而且从程序员的角度看,高级语言的可理解性比机器语言和汇编语言都要强得多,编写程序的效率更高,程序的可读性和可维护性更好。C、C++和Java等都是目前广泛使用的高级语言。

1.3 程序设计方法

1.3.1 结构化程序设计

结构化程序设计(structured programming,SP)方法是由E. Dijkstra等人于1972年提出的,它建立在Bohm和Jacopini证明的结构定理的基础上。结构定理指出:任何程序逻辑都可以用顺序、选择和循环3种基本结构(见图1-3)来表示。

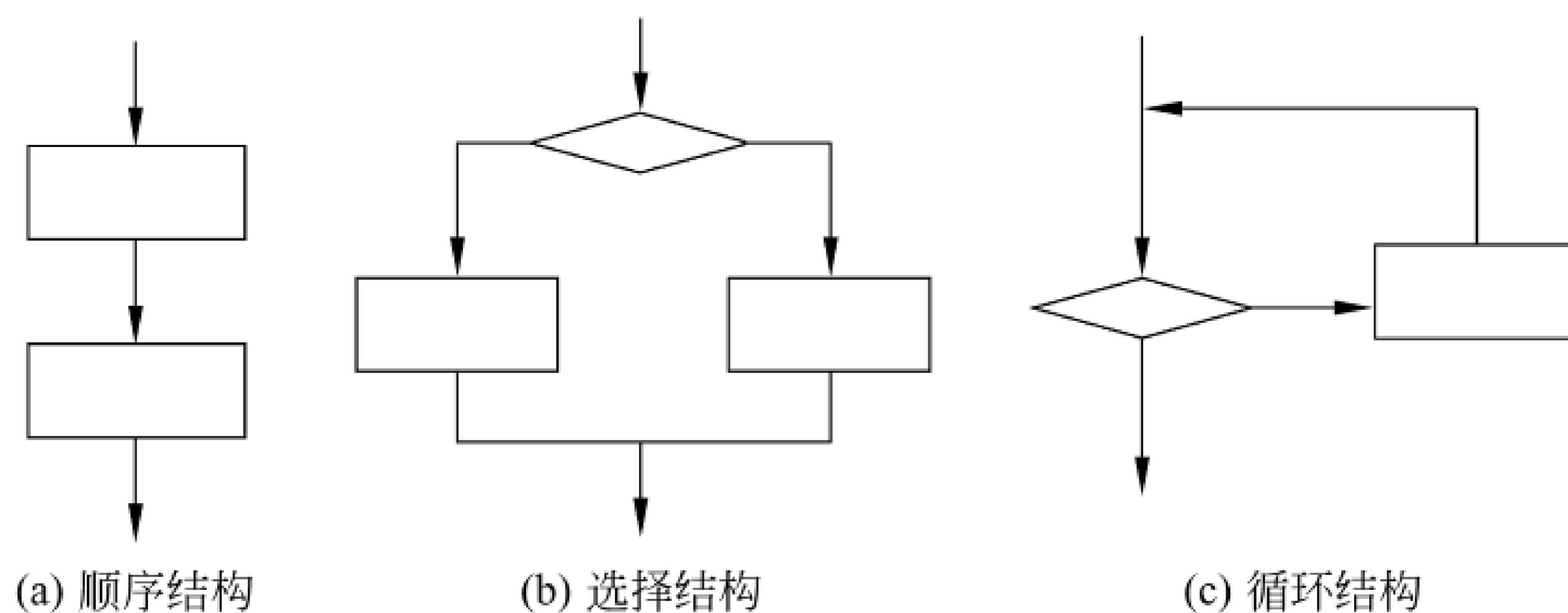


图1-3 程序基本控制结构

在结构定理的基础上,Dijkstra主张避免使用goto语句(goto语句会破坏这3种结构形式),而仅用上述3种基本结构反复嵌套来构造程序。这样,可以使程序的动态执行和静态正文的结构趋于一致,从而使程序易于理解和验证。

程序的基本数据结构有基本数据类型和复合数据类型两类。复合数据类型由基本数据类型按复合规则构成,从而清晰地描述出各种数据结构,并设计出相应的程序。

按照结构化程序设计的要求,程序在设计中应当采用“自顶向下,逐步求精”和“模块化”

原则，在开发大型程序时更是如此。通常，为了设计一个复杂程序，首先需对问题本身作出确切描述，并对问题解法作出全局性决策，把问题分解成相对独立的子问题，对每个子问题用抽象数据及其上的抽象操作来描述；然后，再以同样的方式对抽象数据和抽象操作进一步精确化，直到获得计算机能理解的程序为止。这一过程称为“自顶向下，逐步求精”。在这一过程中，要求程序设计必须首先考虑全局，不要一开始就过多追求细节，先从最上层总目标开始设计，逐步使问题具体化。模块化是将一个大的系统按照子结构之间的疏密程度分解成较小的部分，每部分称为模块。分解的原则须使模块之间相对独立，联系较少。提供给模块外部可见的只是抽象数据及其上的抽象操作，隐蔽了实现细节。整个程序是由层次的逐级抽象的诸模块组成。在结构化的设计和程序实现中，模块一般是以“函数”(function)为单位，函数可以将数据作特定的处理来完成特定的工作。在具体程序设计时，应尽量采用典型的基本控制结构(顺序结构、选择结构、循环结构、函数和过程)，避免使用无条件跳转语句。

在结构化程序设计中，数据与处理数据的方法(函数)是相互分离的。这使得对函数的理解变得很难。尤其随着问题复杂度的提高，数据规模和数据类型的空前激增，导致了许多程序的规模和复杂性均接近或达到了用结构化程序设计方法无法管理的程度。

尽管结构化程序设计技术具有诸多优点，如对于规模较小的软件，结构化程序设计非常适用。但是，当软件规模大到一定程度，结构化程序设计方法就显现出稳定性低、可修改性和可重用性差的弊端。为了克服这些困难，出现了面向对象程序设计方法。

1.3.2 面向对象程序设计

面向对象的程序设计是在结构化程序设计基础上发展而来的另一种重要的程序设计方法，它能够有效地改进结构化程序设计中存在的问题。面向对象的程序与结构化的程序不同，由 C++ 编写的结构化的程序是由一个个函数组成的，而由 C++ 编写的面向对象的程序则是由一个个对象组成的，对象之间通过消息而相互作用。

在结构化的程序设计中，要解决某个问题，就要确定这个问题能够分解为哪些函数，数据能够分解为哪些基本的类型，即思考方式是面向机器实现的，不是面向问题的结构，需要在问题结构和机器实现之间建立联系。面向对象的程序设计方法的思考方式是面向问题的结构，它认为现实世界是由对象组成的，而问题求解的方法与现实世界是对应的。因此，采用面向对象的程序设计方法来解决某个问题，则要确定这个问题是由哪些对象组成的，这些对象之间是如何相互作用的。

面向对象程序设计，是通过为数据和代码建立分块的内存区域，以便提供对程序进行模块化的一种程序设计方法，这些模块可以被用作样板——类，在需要时将其实例化成对象。面向对象的程序设计有 3 个主要特征，即封装、继承和多态性。因此，面向对象程序设计方法要求语言必须具备抽象、封装、继承和多态性等关键要素。本书的后半部分将详细介绍面向对象的程序设计方法。

简单地说，面向对象的程序设计方法可以分成以下 4 个步骤。

① 找出问题中的对象和类。

② 确定每个对象和类的功能，如具有哪些属性、提供哪些方法等。

③ 找出这些对象和类之间的关系，确定对象之间的消息通信方式、类之间的继承和复

合等关系。

④ 用程序代码实现这些对象和类。

虽然面向对象程序设计是在结构化程序基础上发展而来的,甚至许多面向对象程序设计语言还保留了很多结构化的特征,但从上述描述可以看出,面向对象程序设计与结构化程序设计在思路上是明显不同的。结构化程序设计是将问题进行分解,然后用许多功能不同的函数来实现,数据与函数是相互分离的,程序通过函数之间的相互调用来完成功能;而面向对象程序设计是将问题抽象成若干类,将数据与对数据的操作封装在一起,各个类之间可能存在着继承关系,对象是类的实例,程序是由对象组成的,通过对象之间相互传递消息、进行消息响应和处理来完成功能。面向对象的程序设计可以较好地克服结构化程序设计存在的问题,可以开发出健壮、易于扩展和维护的应用程序。

1.4 C语言与C++语言

1.4.1 C语言

1. C语言的历史

C语言是目前世界上最流行、使用最广泛的高级程序设计语言之一。C语言起源于名为ALGOL 60的程序设计语言,基于该语言发展出CPL(combined programming language)语言。在CPL基础上,1967年由Martin Richards为编写操作系统软件和编译器而开发了BCPL(basic combind programming language)语言。1970年美国AT&T公司贝尔实验室的Ken Thompson将BCPL语言改良成B语言,并用B语言在DEC PDP 7计算机上实现了第一个UNIX操作系统。而这几种语言就是C语言的前身。C语言正式的产生是在1972年,由Dennis Ritchie和Ken Thompson在贝尔实验室设计UNIX系统时,共同设计开发出了C程序设计语言。

C语言自问世以来,就是作为UNIX操作系统的开发语言而闻名于世。事实上,目前许多UNIX和Linux操作系统都是用C语言编写的。对于操作系统和系统实用程序以及需要对硬件进行操作的场合,用C语言明显优于其他高级语言。此外,C语言也成为开发各种应用程序的主要工具之一。

随着微型计算机的日益普及,出现了许多C语言版本。由于没有统一的标准,使得这些C语言之间出现了不一致的地方。为了改变这种情况,1988年美国国家标准化协会(ANSI)为C语言制定了一套ANSI标准,成为现行的C语言标准。

2. C语言的特点

归纳起来,C语言具有下列特点。

(1) 数据类型丰富。C的数据类型有整型、实型、字符型、数组类型、指针类型、结构体类型、联合类型等,能用来实现各种复杂的数据类型的运算,并通过引入指针概念,使程序效率更高。另外,C语言具有强大的图形功能,且计算功能、逻辑判断功能强大。

(2) 运算符丰富。C的运算符包含的范围很广泛,共有34个运算符,C语言把括号、赋值、强制类型转换等都作为运算符处理。因此,C的运算类型极其丰富,表达式类型多样化,灵活使用各种运算符可以实现在其他高级语言中难以实现的运算。

(3) 支持结构化程序设计。C是结构化语言,其显著特点是代码与数据的相互分离,即

程序的各个部分除了必要的信息交换外彼此独立。这种结构化方式可使程序层次清晰，便于使用、维护和调试。C语言通过函数为程序员提供了模块化的机制，函数可以方便地进行定义和调用，并提供顺序语句、循环语句、条件语句等机制来控制程序流向，提高程序结构化的程度。

(4) C语言方便、灵活。它把高级语言的基本结构和语句与低级语言的实用性有机结合起来。C语言可以像汇编语言一样对位、字节和地址进行操作，为程序员提供了非常灵活的编程手段。C语言甚至允许直接访问物理地址，可以直接对硬件进行操作。所以，C成为编写系统软件的首选语言。

(5) C语言程序生成代码质量高、程序执行效率高。一般只比汇编程序生成的目标代码效率低10%～20%。

(6) C语言可移植性好。C语言突出的优点就是适合于多种操作系统，如UNIX、AIX、Windows等，也适用于多种机型。

1.4.2 C++语言

C++和C语言相同，也是诞生在贝尔实验室。几乎在ANSI C的委员会致力于建立标准C语言的同时，Bjarne Stroustrup博士在贝尔实验室就将面向对象的思想引入了C语言，并将这种扩展后的语言命名为C++。所以，C++是C语言的扩展，是C语言的一个超集。根据Stroustrup的说法，C++是一个“更好的C语言”。它是一种混合型的语言，既支持传统的结构化程序设计，又支持面向对象程序设计。

C++设计的初衷是为了扩充C语言并引入面向对象程序设计思想。C语言虽然有其强大的地方，但是作为一种结构化编程语言，当程序量相当大的时候，其局限性不可避免地暴露出来。而C++以其对面向对象程序设计方法的支持，成为设计和开发大规模软件的强有力工具。同时，C++在设计时充分考虑了与C语言的兼容性，使得大量基于C语言的开发工作得以继承和发展。许多用C语言编写的代码不需修改就可为C++语言所用，而且原来用C语言编写的众多库函数和实用程序也可以用于C++语言中。因此，对于传统财富不是完全抛弃，而是继承并发展，这是C++语言成功的重要原因。

C++和C的主要区别并非是C++对C语言语法的扩充，而是其对数据抽象和面向对象程序设计方法的支持。C++允许数据抽象，支持封装、继承和多态性等特征。除此之外，C语言程序的设计一般采用自上而下、逐步求精的方式进行软件开发，而C++则是兼有自下而上和自上而下两种方式。目前，C++语言已被广泛用于程序设计的众多领域。实践证明，C++尤其适用于大、中型软件的开发。

C++程序是由类(class)和函数(function)组成的。C++还有一个标准库(C++ standard library)，是由一组函数、常量、类和对象等构成的集合，提供了与操作系统进行交互的基本功能以及经常使用的一些标准类、对象和算法。它们可以在不同的程序中重复使用。因此，要能熟练运用C++进行程序设计，一方面要掌握C++语言本身，二是懂得如何利用C++标准库中现有的类和函数。

1.5 C++编程简介

1.5.1 C++编程的典型过程

1. C++集成开发环境

C++的流行使得许多软件厂商都提供了自己的C++集成开发环境(integrated developing environment,IDE)。著名的有Borland公司的C++ Builder、IBM公司的Visual Age For C++、Microsoft公司的Visual C++等。其中,Visual C++ 6.0及其后续的Visual Studio是当今Windows操作系统下最流行的C++集成开发环境之一。本书的程序实例均用Visual C++ 6.0调试通过。通常来说,一个集成开发环境是将源程序的编写、编译、连接、调试、运行以及应用程序的文件管理等功能有机地集成在一起。

2. 开发C++程序的步骤

开发C++程序通常要经过6个阶段:编辑、预处理、编译、连接、装入和执行。

① 编辑(edit)。这是用编辑器程序来完成的。程序员用编辑器输入C++程序,并进行必要的修改,然后将程序命名存放在磁盘中。C++程序的文件名通常以cpp、cxx、c为扩展名。

② 预处理(preprocess)。在C++系统中,预处理程序是在编译器翻译阶段开始之前自动执行的。C++预处理程序完成对"预处理指令"的处理。预处理指令表示程序编译之前要进行的某些处理操作。这些处理操作指令通常放在要被编译的文件中,如包含其他文件的包含指令、各种文本替换指令等。

③ 编译(compile)。程序员发出编译命令,编译器将C++程序翻译为机器语言代码(也称为目标代码)。

④ 连接(link)。C++程序常常会引用定义在其他程序模块中的数据或函数,如标准库中或特定项目中程序员使用的库中的函数。因此,C++编译器产生的目标代码通常会缺少这部分内容。连接器可以把目标代码和这些函数的代码连接起来产生可执行程序的映像。

⑤ 装入(load)。程序在执行之前,必须先将其装入内存,这是由装入器来完成的。装入器从磁盘中取出可执行程序的映像文件,并把它装入内存的特定位置。

⑥ 执行(execute)。计算机在CPU的控制下执行该程序。

C++程序的编译和连接过程如图1-4所示。

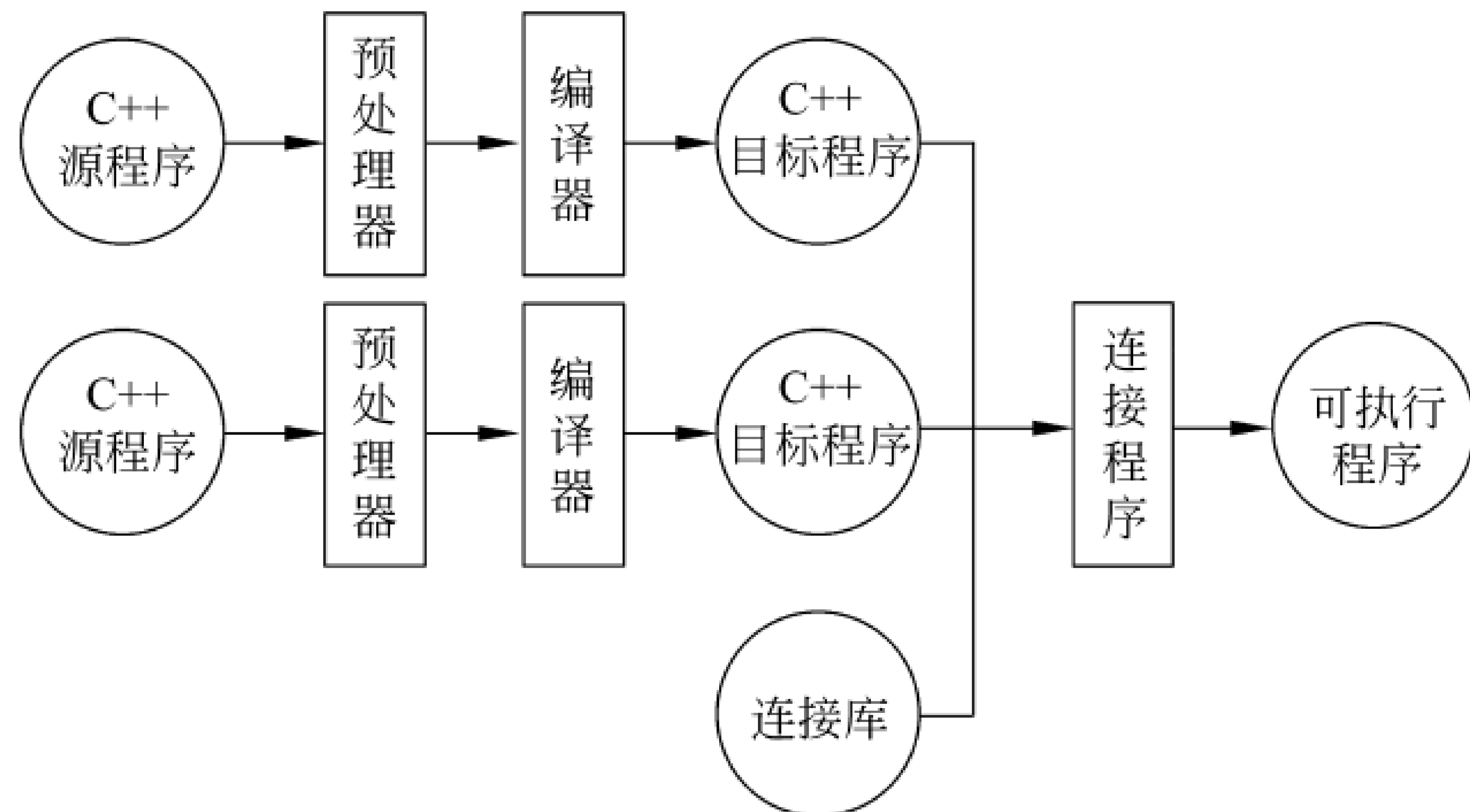


图1-4 C++程序的编译和连接过程

1.5.2 一个简单的 C++ 程序

下面介绍一个简单程序,这个程序包含了构成 C++ 语言程序的基本成分——输入、输出、定义变量及注释等语句。通过本节的学习,可以对 C++ 语言有一个初步认识。

【例 1-2】 编写一个 C++ 程序,输入用户的姓名,然后在屏幕输出“HELLO,***!”。

```
1   //ex1_2.cpp: 输入姓名,然后在屏幕输出问候信息
2   #include<iostream.h>
3   int main()
4   {
5       char name[20];
6       cout<<"Please input your name: ";        //输出提示信息
7       cin>>name;                               //输入姓名
8       cout<<"HELLO,"<<name<<"!\n";             //输出信息,并换行
9       return 0;                                //指示程序运行
10  }
```

这个程序演示了 C++ 语言的几个重要特性。为了介绍的方便,程序中每行的开始都标出了行号,在实际的编程中是不需要也不能有这些行号的。下面详细介绍该程序。

第 1 行是注释语句。该注释语句用来说明整个程序的功能。注释用来对程序进行注解和说明。虽然注释不是程序的有效部分,对于程序的执行不起任何作用,但是给程序添加合适的注释是一种良好的程序设计风格,注释应该视为程序的一个重要组成部分。在 C++ 程序中,注释有两种方式:一是以//开头,表示该行的后续部分都是注释;另一种即 C 语言原有的注释方式,以/*开头,以*/结尾,二者之间的所有字符都是注释。

第 2 行语句是一条预处理指令,它告诉预处理器要在程序中包括输入输出流头文件 iostream.h 的内容,该文件包含了与输入输出流相关的类、类型和函数的说明。如果程序中要使用这些输入输出流功能,必须包括这个文件。

第 3 行语句是每个 C++ 程序都包含的语句。main 后面的括号表示它是一个函数。C++ 程序由一个或多个函数组成,其中有且只有一个 main 函数。即使 main 不是程序中的第一个函数,C++ 程序都是从函数 main 开始执行的。main 左边的关键字 int 表示函数 main 返回一个整数值。

第 4 行的左花括号{与第 10 行的右花括号}表示函数体的开头和结尾。

第 5 行语句定义一个字符数组变量 name,用来存放输入的姓名,最多可以存放 19 个字符。

第 6 行语句是一个输出语句,告诉计算机把引号之间的字符串送到标准输出设备(屏幕)上。

第 7 行语句是一个输入语句,告诉计算机从标准输入设备(键盘)接收数据,并存入后面的变量 name 中。C++ 中的输入和输出是通过字符流来完成的。实际上操作符<<和>>形象地说明了数据的流动方向。在 C++ 中,数据被看成是按照一定的方向流动,这就是“流”这个名称的由来。

第 8 行语句也是一个输出语句,它分成 3 段完成:先输出字符串“HELLO,”,然后输出

变量 name 中的值,最后再输出字符串"!\n"。其中,字符\n 不会在屏幕上显示,它是一个转义符,表示一个换行符,光标移到屏幕中的下一行开头。

第9行语句是函数的返回语句,一般是函数的最后一条可执行语句,返回程序执行的结果。main 函数末尾使用 return 语句时,数值0表示程序顺利结束。

1.6 程序设计风格

所谓程序设计风格,是指借助于好的设计方法编写结构好的程序。在编写源程序时,往往需要采用各种措施来提高程序的可读性、可理解性和可修改性,以利于程序的查错、测试、维护修改及交流。程序设计风格已成为程序员的基本素养之一,对初学的程序员必须加强这方面的训练与培养。

C++是一个复杂的程序设计语言。为帮助读者养成良好习惯,编写更清晰、更易读易懂、更易维护、更易测试和调试的代码,本节针对 C++程序设计,简单介绍几种基本的风格。读者从编写第一个程序开始,就应该遵循一定的程序设计风格,通过长期的训练和积累,使得良好的程序设计风格成为习惯。

1. 以简洁明了的方式编写 C++程序

以简洁明了的方式编写 C++程序是一种良好的程序设计习惯。通常称这种编写程序的方法为 KIS(keep it simple,保持简洁),不要使用不常用或稀奇古怪的方式来编写程序。

2. 缩排规则

所谓缩排规则就是使程序的书写格式应能较好地反映出该程序的层次结构。例如,处于同一层的语句都从同一个字符位置开始书写;将每个函数的整个函数体在定义函数体的花括号中缩排一级,可使程序中的函数结构更明显,使程序更易读;确定一个自己喜欢的缩排长度,以后一直坚持这个缩排长度。

3. 标识符命名规则

标识符的命名虽然可由程序员任意选择,但最好选择能够反映相关功能和特征的单词来命名,以便于见名识意。例如,选择有相对意义的变量名,可使该程序比较容易理解;程序中使用的常量,可根据其意义,在常量说明中起一个有助于记忆的名字;避免用下划线和双下划线开头的标识符,因为 C++编译器内部使用这类名称。本书采取表 1-1 的标识符命名规则。

表 1-1 标识符命名规则

标识符类型	命名规则	例子
类	类名一般是一个名词或名词词组,要能反映该类的意义或功能,采用大小写混合的方式,每个单词的首字母大写	class Controller; class Student; class EmployeeInformation;

函数或方法	一般是一个动词或动词词组,要能反映该函数或方法的功能,采用大小写混合的方式,第一个单词的首字母小写,其后单词的首字母大写	display();
		diaplayMessaget();
		getStudentNum();

续表

标识符类型	命 名 规 则	例 子
变量或成员变量	采用大小写混合的方式,第一个单词的首字母小写,其后单词的首字母大写。变量名不应以下划线或美元符号开头,尽管这在语法上是允许的。变量名应简短并能反映变量的意义。尽量避免单个字符的变量名,除非是一次性的临时变量	float width char c; int i;
常量	常量名中的字母应该全部大写,单词间用下划线隔开	#define MIN_WIDTH 10 #define MAX_WIDTH 255

4. 注释

注释是增加可读性、可理解性的常用措施。注释虽然不是程序的有效部分,但是,给程序添加合适的注释是一种良好的程序设计风格,注释应该视为程序的一个重要组成部分。在 C++ 程序中可以使用单行注释与多行注释。下面是 3 种注释的例子。

(1)

```
//Handle the condition
```

(2)

```
/*
 * Here is a block comment.
 */
```

(3)

```
if (a==2) {
    return 1;                    /* special case */
} else {
    return isPrime(a);           /* works only for odd a */
}
```

5. 注意大小写英文字母

C++ 语言中是区分英文字母的大小写的。因此,a1 和 A1 是两个不同的标识符。

6. 输出信息

利用 C++ 的输出语句尽量将要输出的信息组织得直观清晰、布局合理,使之形象化、表格化、页面化、自动成文,便于他人看懂和存档。

上述这些原则仅是养成良好的程序设计风格的基本要求。事实上,许多具有丰富程序设计经验的专家总结出了很多详细、实用、经典的编程规范,读者可以选择合适的规范加以实践。程序设计风格的原则是非常简单和明确的,而最重要也是最难做到的就是在实践中始终如一地按照这些风格和规范进行程序设计。要想成为高水平的程序设计员,良好的程序设计风格是必备的素质。

习 题 1

1.1 指出下列项目哪些是硬件,哪些是软件。

(1) CPU　　(2) C++ 编译器　　(3) ALU

(4) C++预处理器　　(5) 输入单元　　(6) 编译程序

1.2 为什么面向对象程序设计方法成为当前软件开发的主流方法?

1.3 简述C++程序开发的步骤。

1.4 模仿本章中的两个程序,编写下列C++语句。

(1) 打印信息“Enter two numbers”。

(2) 将输入的两个数读入变量a与b中。

(3) 将变量a和b的乘积赋给变量c。

(4) 打印“a * b=”,加上变量c的值。

1.5 判断下列变量名是否体现了良好的风格,并说明理由。

m829123、t5、j7、her_sales、his、account_total、a、i、z、_under_bar、_name、sumOfScore、averageage。

1.6 从Internet上寻找一些编写程序的规范,从中选择一个合适的作为今后程序设计风格规范。

第 2 章 数据类型、运算符与表达式

【学习内容】

本章介绍C++描述数据的机制——数据类型，以及对数据进行处理的最基本的手段——运算操作和表达式。内容包括：

- ◆ C++语言的基本元素、字符集和关键字。
- ◆ 基本数据类型的表示方法。
- ◆ 常量和变量。
- ◆ 类型转换。
- ◆ 各种运算符与表达式。

【学习目标】

- ◆ 理解字符集、标识符和关键字的意义。
- ◆ 掌握基本数据类型的定义以及不同数据类型之间的转换机制。
- ◆ 掌握各种常量的性质和定义。
- ◆ 掌握变量声明的方法。
- ◆ 熟练运用各种运算符与表达式。

2.1 C++的字符集和关键字

从形式上看，程序只不过是一个字符序列。对于C++程序来说，能够出现在程序中的字符是有限的，即C++程序是由一定字符集中的字符构成的。

2.1.1 字符集

字符是一些可区别的符号，C++语言规定了构成源程序的基本源字符集(basic source character set)，包括下列字符。

(1) 大小写的英文字母：A～Z，a～z。

(2) 数字字符：0～9。

(3) 其他字符：_ { } [ ] # ( ) < > % : ; . ? * + -/ ^ & | ~ != , \ " '。

(4) 编辑性字符：空格(space character)，水平跳格符(horizontal tab)，垂直跳格符(vertical tab)，换页符(form feed)，换行符(newline)。

2.1.2 标识符

在高级程序设计语言中,各种数据对象都是有名字的,它们都是用所谓的标识符来命名。一般说来,标识符是以非数字字符开头的、由字母和数字组成的有穷字符序列,通常用来表示常量、变量、语句标号以及用户自定义函数的名称。不同的语言对于标识符的组成形式的规定也各不相同。C++标识符的定义很灵活,但作为标识符必须满足以下规则。

(1) 所有标识符必须由一个字母(a～z,A～Z)或下划线(_)开头。

(2) 标识符的其他部分可以用字母、下划线或数字(0～9)组成。

(3) 大小写字母是区分的,表示不同的符号。

(4) 标识符只有前32个字符有效。

(5) 用户定义的标识符不能与C++的关键字相同。

例如,smart、_decision、key_board、FLOAT是正确的标识符;而5smart、bomb?、key.board都不是合法的标识符,而int、float虽然是合法的标识符,但不是合法的用户定义标识符,它们是关键字。

2.1.3 关键字

关键字是C++语言预定义的具有特殊意义的标识符。程序员不能将关键字挪作它用,只能按照C++语言为它们定义的意义来使用,所以关键字也称为保留字。有些关键字代表计算机的动作,有的表示语言预定义的某种数据类型,有的用于标识某个程序段。表2-1列出了C++的关键字。关于C++关键字的意义和用法将在后面介绍。

表2-1 C++的关键字

asm	do	if	return	typedef
auto	double	inline	short	typeid
bool	dynamic_cast	int	signed	typename
break	else	long	sizeof	union
case	enum	mutable	static	unsigned
catch	explicit	namespace	static_cast	using
char	export	new	struct	virtual
class	extern	operator	switch	void
const	false	private	template	volatile
const_cast	float	protected	this	wchar_t
continue	for	public	throw	while
default	friend	register	true	
delete	goto	reinterpret_cast	try	

另外,第1章曾提到预处理功能,C++还提供了一些预处理指令(参见5.9节),如#define、#error、#elif、#endif、#ifdef、#ifndef、#undef、#line和#pragma等。显然,#之后都是一些合法的标识符。虽然C++未将它们列入关键字加以保护,程序员可以在程序设计时使用它们作为自定义标识符表示其他的意义,但是因为它们已经具有特殊的意义,为了避免伤害程序设计风格,仍然不提倡将这些标识符作为自定义标识符加以使用。

2.2 基本数据类型

如前面所述,程序从本质上来说是描述一定数据的处理过程。高级语言通过数据类型来描述程序中的数据对象,C++ 语言也提供了丰富的预定义数据类型以及相关的运算,它们和语句组合在一起,可以方便地表达复杂的客观世界。

同一类型的数据,其成分都具有相同的特性,可进行相同的操作。例如,对于整数来说,都有相同的数学特性和相同的内部表示法,对于实数、复数和逻辑数等也是如此。在高级语言程序中,每个数据对象都属于确定的数据类型。数据类型的定义包含两个方面的意义,即该类型所有可能取的值以及在这些值上可容许的操作。类型概念的明显特征可以概括如下。

(1) 类型决定变量或表达式所能取值的集合。

(2) 每一个值属于一个且仅属于一个类型。

(3) 每一种操作要求一定类型的操作数,并且得出一定类型的结果。

(4) 一种类型的值及其规定的基本操作的性质,可由一组公理阐明。

根据数据类型的复杂程度,C++ 提供的数据类型可分为两类:基本数据类型和复合数据类型。这里先介绍 C++ 基本数据类型,包括整型(int)、浮点型(float 和 double)、字符型(char)、布尔型(bool)。复合数据类型将在后面章节讨论。

2.2.1 整型

在 C++ 中,整型数据可以分成有符号和无符号两大类,每类又包含几种类型,各种整型的名字、意义、存储特性和表示数据范围都不相同。需要特别指出的是,在不同的机器、不同的操作系统和不同的 C++ 编译器中,数据类型的表示方式各不相同,因此所能表示的数据范围也不一样。按照 C++ 规定,各种类型的表示范围应该满足条件:短整型≤整型≤长整型。

表 2-2 给出了 Microsoft C++ 中各种整型类型的特性。

表 2-2 C++ 的整型

类别	类 型 名	意 义	存储特性	表示范围
有符号	signed short int(简写为 short)	有符号短整型	字长为 2 字节	−32768～32767
	signed int(简写为 int)	有符号整型	字长为 4 字节	−2147483648～2147483647
	signed long int(简写为 long)	有符号长整型	字长为 4 字节	−2147483648～2147483647
无符号	unsigned short int(简写为 unsigned short)	无符号短整型	字长为 2 字节	0～65535
	unsigned int(简写为 unsigned)	无符号整型	字长为 4 字节	0～4294967295
	unsigned long int(简写为 unsigned long)	无符号长整型	字长为 4 字节	0～4294967295

整型数据可以参与的运算包括算术运算、关系运算、逻辑运算、按位运算等,这些内容将在 2.5 节进行详细介绍。

2.2.2 浮点型数据

浮点型数又称为实型数。C++中有3种浮点类型：float、double和long double。与整型类似，C++也未规定各种浮点类型的表示范围，只是规定各种类型的表示精度应该满足条件：浮点型≤双精度型≤长双精度型。

表2-3给出了Microsoft C++中各种浮点类型的特性。

表2-3 C++的浮点类型

类型名	意　义	存储特性	可表示的最小非0的绝对值	可表示的最大的绝对值
float	浮点型	字长为4字节	1.175494351E－38	3.402823466E+38
double	双精度型	字长为8字节	2.2250738585072014E－308	1.7976931348623158E+308
long double	长双精度型	字长为8字节	2.2250738585072014E－308	1.7976931348623158E+308

注意：浮点数均为有符号浮点数，没有无符号浮点数。

浮点类型数据可以参与的运算包括算术运算、关系运算、逻辑运算等，具体参见2.5节。

2.2.3 字符型数据

字符型用于表示字符数据，以方便字符处理。实际上，C++将字符类型看成是一种整型，并要求字符类型的长度不超过short类型。C++为每个字符数据对应一个整数值，如Microsoft C++中的字符类型的取值集合为ASCII字符集，ASCII为每个合法的字符定义了一个整数，该整数就是该字符在C++的字符类型中的数值表示。C++中有3种字符类型：char、signed char和unsigned char。同样，C++也没有规定字符类型的存储大小，而只是规定字符类型要能够足够表示所有的字符数据。char、signed char和unsigned char占用的字节数是相同的，字符类型数据的每一位都用于表示数据的值，其中signed char的值是有符号的，unsigned char的值是无符号的，char类型的值是有符号还是无符号取决于C++的具体实现，C++没有统一的规定。表2-4给出了Microsoft C++中字符类型的特性。

表2-4 C++的字符类型

类型名	意　义	存储特性	表示范围
char	字符型	字长为1字节	－128～127
signed char	有符号字符型	字长为1字节	－128～127
unsigned char	无符号字符型	字长为1字节	0～255

字符在计算机中以其ASCII码方式表示，其长度为1个字节，有符号字符型数取值范围为－128～127，无符号字符型数取值范围是0～255。因此，在C++语言中，字符型数据在操作时可按整型数处理，如果某个变量定义成char，则表明该变量是有符号的，即它被转换成有符号的整型数。

由于字符类型数据可以作为整型数据参与运算，所以字符类型可以施加的运算与整型完全一样。

2.2.4 布尔类型

布尔类型用于表示布尔逻辑数据，布尔逻辑数据只有两个：true 和 false。在 C++ 中，布尔型的数据可以作为整型数据进行运算，true 为非 0 整数，false 为整数 0；整型数据也可以作为布尔型数据进行运算，非 0 整数为 true，整数 0 为 false。

布尔类型数据上可以施加逻辑运算，参见 2.5 节。

2.3 常量与变量

2.3.1 常量

在 C++ 程序运行中，有一些数据是一直不变的，这些数据称为常量。下面分别介绍各种类型的常量。

1. 整型常量

说明：本书没有给出 C++ 语言完整、准确的语法定义，只是对于其中常用的、基本的语言成分的语法进行了说明，更为准确的定义可参见参考文献[1]。本书的大多数语法描述采取了扩充的巴克斯范式的形式，每个语法单位用尖括号括起来，其定义都由一组规则组成，每个规则说明了该语法单位的构成规定，即由哪些语法单位和符号构成。被定义的语法单位后面标有箭头“→”，读为“定义为”。一个语法单位可以有多种定义形式，各种定义之间用竖线“|”分隔，“|”读为“或”。另外，还用到了下面几个符号。

(1) 用$\{\alpha\}$表示 α 可以重复任意多次(包括 0 次，即不出现)。

(2) 用$\{\alpha\}_0^n$ 表示 α 可任意重复 0～n 次。

(3) 用[α]表示$\{\alpha\}_0^1$，即表示 α 的出现可有可无。

例如：

<整型常量>→
<十进制整型常量> [<整型后缀>] |
<八进制整型常量> [<整型后缀>] |
<十六进制整型常量> [<整型后缀>]
<八进制整型常量>→ 0<八进制数字> {<八进制数字>}

说明：“整型常量”这一语法单位可以由十进制整型常量、八进制整型常量和十六进制整型常量构成，这些常量的后面可以带有整型后缀，也可以没有整型后缀，而八进制整型常量则由 0 开头，后面可以是八进制数字组成的任意串。

C++ 中整型变量的语法如下：

<整型常量>→<十进制整型常量> [<整型后缀>] |
<八进制整型常量> [<整型后缀>] |
<十六进制整型常量> [<整型后缀>]
<十进制整型常量>→0|<非 0数字> {<数字>}
<八进制整型常量>→0 <八进制数字> {<八进制数字>}
<十六进制整型常量>→0x<十六进制数字> {<十六进制数字>} |

```
                 0X<十六进制数字>{<十六进制数字>}
<非 0数字>→1|2|3|4|5|6|7|8|9
<数字>→0 |<非 0数字>
<八进制数字>→0|1|2|3|4|5|6|7
<十六进制数字>→0|1|2|3|4|5|6|7|8|9 |
                     a|b|c|d|e|f|A|B|C|D|E|F
<整型后缀>→<无符号整型后缀>[<长整型后缀>]|
                 <长整型后缀>[<无符号整型后缀>]
<无符号整型后缀>→u|U
<长整型后缀>→l|L
```

可见,按不同的进制区分,整型常量可以有 3 种表示方法。

(1) 十进制整数:以非 0 数字开头、由十进制数字组成的整数,如 220、45900。

(2) 八进制整数:以 0 开头、由八进制数字组成的整数,如 06、0106。

(3) 十六进制数:以 0X 或 0x 开头、由十六进制数字组成的整数,如 0X0D、0XFF、0x4e。

此外,可在整型常数后添加一个 L 或 l 字母表示该数为长整型数,如 22L、0773L、0Xae4l。若加上一个 u 或 U 字母表示该数为无符号整型数,如 27u、0400u、0xb8000000u。若加上一个 ul 或 UL 字母表示该数为无符号长整型数,如 27ul、0400UL、0xb8000000UL。

如果整数的值超出相应整型所能表示的范围,称为整数溢出。整数溢出将造成程序逻辑上的错误,要特别注意加以避免。

2. 浮点型常量

C++ 语言表示浮点型常量有两种方法,即十进制表示法和科学表示法。其中,十进制表示法就是人们熟悉的小数表示法,科学表示法也就是指数表示法(即 E 表示法),这两种表示方法的语法如下:

```
<浮点常量>→<尾数部分> [<指数部分>] [<浮点后缀>]  |
                 <数字串> <指数部分> [<浮点后缀>]
<尾数部分>→[<数字串>] .<数字串>   |
                 <数字串>.
<指数部分>→e [<符号>] <数字串>|
                 E[<符号>] <数字串>
<符号>→+ |-
<数字串>→<数字>{<数字>}
<浮点后缀>→f|l|F|L
```

下面是几点说明。

(1) 浮点常数只有一种进制——十进制。

(2) <浮点后缀>可以为 f、l、F 和 L,f 和 F 表示 float,而 l 和 L 表示 long double,<浮点后缀>也可以省略,如果省略,常量的类型为 double,即所有浮点常量都被默认为 double。

（3）绝对值小于 1 的浮点数，其小数点前面的零可以省略，如 0.22 可写为.22，−0.0015E−3 可写为−.0015E−3。

3. 字符型常量

C++ 的字符集中字符可以直接用单引号括起，以表示字符数据，如'a'，'9'，'Z'。字符型常量的语法如下：

<字符型常量>→'<c字符>' |
L'<c字符>'
<c字符>→字符集中字符，但不能是单引号'、反斜线\和换行符(newline character) |
<转义序列>
<转义序列>(escape-sequence)→<简单转义序列> |
<八进制转义序列> |
<十六进制转义序列> |
<简单转义序列>→\'|\"|\?| \\|
\a|\b|\f|\n|\r|\t|\v
<八进制转义序列>→\<八进制数字> |
\<八进制数字><八进制数字> |
\<八进制数字><八进制数字><八进制数字>
<十六进制转义序列>→\x<十六进制数字>{<十六进制数字>}

在 C 和 C++ 中，字符数据是用该字符的 ASCII 码值来表示的，即字符数据的内部表示就是该字符的 ASCII 码值。例如，十进制数 97 表示小写字母 a，十进制数 65 表示大写字母 A，十六进制数 0x41 表示大写字母 A，八进制数 0101 表示大写字母 A。

一些不能用符号来表示的控制符，只能用 ASCII 码值来表示。例如，十进制数 10 表示换行，十六进制数 0x0d 表示回车，八进制数 033 表示 Esc。C++ 中也有另外一种表示方法，例如，'\033'表示 Esc，这里\0 符号后面的数字表示八进制的 ASCII 值，当然这种表示方法也适用于可直接用符号表示的字符。

另外，C++ 中有些常用的字符用以下特殊规定（又称转义符）来表示（见表 2-5）。

表 2-5　常用转义符意义

规　定　符	等　价　于	含　　义	规　定　符	等　价　于	含　　义
'\f'	'\X0C'	换页符	'\\'	'\X5C'	\符
'\r'	'\X0D'	回车符	'\''	'\X27'	'符
'\t'	'\X09'	制表符	'\"'	'\X22'	"符
'\n'	'\X0A'	换行符			

字符数据在内存中是以 ASCII 码值存储，它的存储形式与整型数的存储形式相同。C++ 语言将字符型数据视为一种整型数据。一个字符数据既可以字符形式输出，也可以整数形式输出。以字符形式输出时，需要先将存储单元中的 ASCII 代码转换成相应字符，然后输出；以整数形式输出时，直接将 ASCII 代码作为整数输出。

关于 ASCII 字符集的详细信息参见附录 B 。

4. 字符串常量

有时候一个字符序列也会成为处理的数据,例如一个单词、一句话乃至一段文字等。这种字符序列称为字符串常量,一般用双引号括起来表示。字符串常量的语法如下:

```
<字符串常量>→"[<s字符序列>]"|
              L"[<s字符序列>]"
<s字符序列>→<s字符>{<s字符>}
<s字符>→字符集中字符,但不能是双引号"、反斜线\和换行符(newline character)|
        <转义序列>
```

需要特别指出的是,不要将字符常量与字符串常量混淆,字符常量是指单独一个字符,而字符串常量代表一个有穷的字符序列,也可以是长度为 1 的字符序列,甚至可以是长度为 0 的字符序列。例如:

'a'是字符常量;

"a"是长度为 1 的字符串常量,即由 1 个字符 a 组成的字符串;

""是长度为 0 的字符串常量,即由 0 个字符组成的字符串。

在内部存储方面,字符串常量占用连续的存储单元,字符串常量中的每个字符按照顺序存放在这片连续单元中,在最后的字符后面加一个字符'\0'作为字符串结束标志。

'a' 在内存的存放形式: | a | 。

"a"在内存的存放形式: | a | \0 | 。

""在内存的存放形式: | \0 | 。

5. 布尔型常量

C++ 的布尔型常量很简单,只有 false 和 true,即:

```
<布尔型常量>→ false|true
```

前面已经指出,在 C++ 中,布尔型的数据是作为整型数据进行运算的。

注意:*false 和 true 是布尔型常量,不是字符串。*

2.3.2 变量

C++ 程序中,除了常量以外,大多数数据对象是动态变化的,它们可能随着程序的执行,通过运算不断改变自身的值。例如,将 100 个整数逐步累积求和,这个过程中,累积的“和”就是一个不断变化的量。一般地,将在程序运行过程中其值可以改变的量称为变量。

变量的声明(declarations)

一个变量应该有一个标识符(称为变量名)来标识它,并且在内存中占据一定的存储单元用来存放变量的值。

在 C++ 语言中,要求对所有用到的变量作强制定义,必须先声明后使用。基本数据类型的变量声明形式如下:

```
<变量声明>→[<存储类别描述符>]<类型描述符>  <变量表>
<存储类别描述符>→auto|register|static|extern
<类型描述符>→类型名
<变量表>→<标识符列表>
<标识符列表>→<标识符>{,<标识符>}
```

C++中的变量可以有4种存储类别描述符：auto、register、static和extern。不同的存储类别决定了变量在内存中存在的时间。auto是默认的存储类别。本书将在5.6节详细介绍存储类别。

类型名是指C++的有效数据类型的名字，这些名字都是标识符。变量表是一个或多个标识符名，每个标识符之间用“,”分隔，表示这些变量都具有类型名标识的数据类型，即指明这些变量的取值类型，也规定变量的存储特性。下面是一些变量声明的例子。

```
unsigned char c;              //c被定义为无符号字符变量
int age, number, width;       //age、number和width被定义为有符号整型变量
unsigned long c;              //c被定义为无符号长整型变量
float price, weight;          //price和weight被定义为单精度浮点型变量
double averageScore, sum;     //averageScore和sum被定义为双精度浮点型变量
```

程序设计风格提示：变量名一般是由以小写字母开头能够反映变量意义的英文单词组成，如果需要用多个单词来命名变量，那么第一个单词的首字母小写，后续单词的首字母大写，必要时可以对较长的单词进行缩写，如averageScore、theTotalNum等。

2.3.3 符号常量

在程序运行过程中，其值不能改变的量称为常量，包括整型、实型、字符、字符串等各种类型的常量。C++可以为常量定义一个“名字”来代表这些常量。当用一个标识符代表一个常量时，称之为符号常量。把直接出现在程序中的常量称为文字常量。

C++定义符号常量的格式如下：

```
#define <标识符><常量>
```

#define是C++的预处理指令，说明程序中的<标识符>都代表了<常量>。例如：

```
#define PRICE 30
#include<iostream.h>
int main()
{
    int num, total;
    num=10;
    total=num*PRICE;
    cout<<"total="<<total<<endl;
    return 0;
}
```

这个例子中的 PRICE 就称为符号常量,即凡在程序中出现的 PRICE 都代表 30。

程序设计风格提示: 成为常量的"名字"的标识符要能够反映该常量的意义。即使一个常量代表了不同的意义,也应当为该常量定义多个反映其意义的"名字"。一般情况下,符号常量名用大写字母组成,如果需要用多个单词来描述一个符号常量,那么这些单词之间可以用下划线分隔,如 TOTAL_SCORE 等。程序中应当尽可能减少文字常量的出现次数,尽量多使用符号常量。

2.4 类 型 转 换

C++允许不同类型的数据进行转换,可以将一种数据类型的数据转换成另一种类型的数据,语法如下:

```
(<目标数据类型>)<原数据类型的数据>
```

例如:

(int) 3.14　将 3.14 转换成整数 3。

(double) 3　将 3 转换成双精度浮点数 3.0。

由于各种数据类型在表示范围和精度上是不同的,所以有的转换不会丢失数据的精度,而有的转换会丢失数据的精度。例如,将 int 类型数据转换成 double 类型,一般不会导致数据的改变,而将 double 类型的数据转换成 int 类型,则会截去 double 的小数部分,从而可能改变数据的值;与此类似,将大的整数类型变为较小的整数类型,如 long 转换成 short,也可能改变数据的值。为此,C++规定了一个"提升规则",说明如何保证当一种数据类型转换为另一种数据类型时不会丢失数据的精度。C++按照各种数据类型的表示范围和精度,将各种数据类型由高到低做了排序,如表 2-6 所示。

表 2-6　数据类型的表示范围和精度级别

数 据 类 型	
long double	高
double	
float	
unsigned long int	
long int	
unsigned int	
int	
unsigned short int	
short	
unsigned char	
char	低

将数值转换成较低的类型可能导致数值不正确。所以,C++要求,如果要将数值转换成较低的类型,必须显式地使用强制类型转换;如果将数值转换成较高的类型则可以通过隐式的类型转换,下节将介绍的表达式中的类型转换就属于这种情况。

类型转换是指把一种类型的数据转换成另一种类型的数据。类型转换有两种:隐式类型转换和强制类型转换。

1. 隐式类型转换

隐式类型转换由系统自动隐含地进行。当表达式中操作数据的类型不同时,要进行隐式类型转换,使表达式中的数据类型相同。例如,在算术表达式和赋值表达式中类型不同时,就进行隐式类型转换。

算术表达式中隐式类型转换规则如下。

(1) 表达式中如有 char、short 和 enum 类型的数据时，自动将其转换成 int 类型。

(2) 把表达式中不同类型的数据转换成精度最高、占用内存最多的那个数据的类型。

在赋值表达式中，自动将赋值运算符右边表达式的值的类型转换成左边变量的类型。这时如果左边变量类型的精度低于右边表达式值的类型时，可能会丢失数据的精度。

2. 强制类型转换

强制类型转换又称显式类型转换。形式为：

```
(<类型>)<表达式>
```

或者

```
<类型>(<表达式>)
```

它把表达式值的类型强制转换成指定的类型。例如：

```
(double)3/2
```

或者

```
double(3)/2
```

先把整数 3 强制转换成双精度类型，再把 2 隐式转换成双精度类型，最后得到的值是双精度数 1.5。如果把上式写成：

```
(double) (3/2)
```

或者

```
double(3/2)
```

则先计算 3/2 得到整数值 1，再把 1 转换成双精度类型 1.0。

强制类型转换也是一种运算符，属于一元运算符，有较高的优先级和自右至左的结合特性。

在类型转换(包括隐式转换和显式转换)中，由精度低、占用内存少的数据类型转换成精度高、占用内存多的数据类型时，仅改变类型，不改变它的值；反之，不仅改变类型，也可能改变其值，使用时要注意。

2.5 运算符和表达式

C++ 提供了丰富的运算符(操作符)以实现各种运算功能，主要包括算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符等。除此之外，还有一些用于完成特殊任务的运算符。这些运算符和运算量(也称为操作数)一起构成了表达式，程序通过计算表达式完成对数据的处理。在第 5 章中还会看到表达式可以包括函数调用，即把函数调用的返回值作为运算量参与运算。附录 A 给出了各种运算符的计算优先级和结合性质，计算机计算表达式时总是按照该表的规定进行的。运算符的优先级规定了不同运算符出现在同一表达式中

优先运算的级别,而结合性质则规定了同等优先级的运算符出现在同一表达式中运算的顺序。其中,右结合是指对于同等优先级的运算符从右至左依次计算,而左结合则表示从左至右进行计算。优先级和结合性质一起决定了计算的先后次序。

根据组成表达式的运算符的不同,表达式可以分为算术表达式、关系表达式、逻辑表达式、位运算表达式、逗号表达式和赋值表达式等。

2.5.1 算术运算

1. 基本的算术运算

基本的算术运算符主要有:

+ 加法。运算结果为两个操作数的和。

- 减法。运算结果为左操作数减去右操作数的差。

* 乘法。运算结果为两个操作数的乘积。

/ 除法。运算结果为左操作数除以右操作数的商。

% 求余。运算结果为左操作数除以右操作数的余数。

这5种运算符都是二元运算符,要求有两个操作数,即左操作数和右操作数,构成表达式的语法为:

```
<算术表达式>→<常量>|
              <变量>|
              <操作数><二元算术运算符><操作数>|
              (<算术表达式>)
<操作数>→<表达式>
```

可见,单独的常量和变量是表达式,任何表达式又可以成为操作数。表达式和运算符可以构成更复杂的表达式。在表达式中,运算符之间的优先关系与数学上的规定类似。

(1) 括号中的表达式优先级最高,有括号嵌套时,内层括号优先于外层括号,由内层向外层求值。

(2) 乘法、除法和求余运算优先级次之,三者的优先级相同。如果表达式中有连续的乘法、除法和求余运算,则遵循左结合原则,从左算到右。

(3) 加法和减法优先级较低,二者优先级相同。如果表达式中有连续的加法和减法运算,也遵循左结合原则,从左算到右。

数据类型包括两方面的含义:值的集合和数据上的运算。上述运算符中,+、-、*、/可以施加于整数和浮点类型上,分别代表整数和实数上的加、减、乘、除运算,其结果类型与操作数的类型相同。求余运算%只用于整型数据,其结果也是整型。注意,/用于整数和实数的含义是不一样的(即整数和实数的除法运算都用同一个符号/来表示,这种现象称为重载)。两个实数作/运算,结果为实数;两个整数作/运算,结果为整数,其具体结果取决于C++在具体机器上的实现:一般来说,大多数实现都采取“向零取整”,即直接截去商的小数部分。例如:

5/3　　结果为1。

－5/3　结果为－1。

5/2　　结果为2。

－5/2　结果为－2。

此外，整型数据的求余运算%需注意，其运算结果在不同C++的实现中也不一样。C++规定，如果两个操作数都是非负的，那么结果一定是非负的；否则，要看C++的具体实现，C++标准对这种情况未作明确规定，如在Microsoft C++实现中，求余运算%的结果总是与左操作数的符号一致。这样，整除运算和求余运算的关系为：

e1%e2＝e1－(e1/e2)*e2

下面是一些例子：

11/3	结果为3。	11%3	结果为2。
－11/3	结果为－3。	－11%3	结果为－2。
11/－3	结果为－3。	11% －3	结果为2。
－11/－3	结果为3。	－11%－3	结果为－2。

对于＋、－、*、/来说，如果两个操作数的类型不同，C++会对操作数作隐式的类型转换，使得两个操作数具有相同的数据类型。具体做法是，总是将类型较低的操作数转换成较高的数据类型，运算结果也具有较高的数据类型。例如：

4/2　　结果为整型的2。

4/2.0　先将4转换成double类型，结果为double类型的2.0。

2. 其他算术运算

除了上面介绍的二元运算符，C++还有一元运算符，这些运算符只有一个操作数。例如，取正运算＋和取负运算－，其语法如下：

```
<算术表达式>→+<操作数>|
              -<操作数>
```

一元取正运算＋和二元加法运算的符号是相同的，取正运算的结果就是操作数本身，而一元取负运算的结果则是操作数的负数。

C++还有两个使用起来非常方便、灵活的算术运算符：自增运算符＋＋和自减运算符－－。这两个运算符都是一元运算符，只有一个操作数，而且该操作数必须具有左值。其形式如下：

```
<算术表达式>→<操作数>++|
              ++<操作数>|
              <操作数>--|
              --<操作数>
```

它们的功能相对复杂一些，而且＋＋和－－出现在操作数之前和之后具有不同的功能，具体见表2-7。

表 2-7　算术运算符++和--的功能

运算符	名　称	示　例	说　明
++	前自增	++ a	将 a 加 1,a 增加后的新值为运算的结果
++	后自增	a ++	将 a 加 1,a 增加前的旧值为运算的结果
--	前自减	-- a	将 a 减 1,a 减少后的新值为运算的结果
--	后自减	a --	将 a 减 1,a 减少前的旧值为运算的结果

自增运算和自减运算的优先级相同,它们都比括号的优先级低,但比加法、减法、乘法、除法和求余运算要高。下面给出几个示例,假设变量 a 当前值为 5，则有：

++ a+12　结果为 18(运算后 a 为 6)。

a +++12　结果为 17(运算后 a 为 6)。

--a*12　结果为 48(运算后 a 为 4)。

a-- *12　结果为 60(运算后 a 为 4)。

2.5.2　关系运算

现实世界中通常需要对数据进行比较,为此 C++ 也提供了一组关系运算符,以实现对数据进行关系比较,包括＜、＞、＜=、＞=、==和!=等,它们分别计算数据的小于、大于、小于等于、大于等于、相等和不相等关系。关系运算的结果反映了操作数的大小关系,为布尔类型,即 true 和 false。关系运算表达式的语法如下：

<关系表达式>→<操作数><关系运算符><操作数>

下面是一些关系运算表达式：

1＜=0　　结果为 false。

x!=x+1　结果总为 true。

x＞y　　结果表示 x 是否大于 y。

注意：对于＜=、＞=、==和!=,这些运算符都是由两个字符组成,书写时两个字符间不能加空格。

2.5.3　逻辑运算

前面的关系运算能够得到布尔型的结果：true 和 false。但是,仅有这些简单的条件测试不能满足现实世界中复杂的逻辑计算问题。例如,当需要判断各种复杂的组合条件时,还必须提供逻辑运算的能力。

C++ 提供了 3 种逻辑运算符：二元逻辑与运算符(&&)、二元逻辑或运算符(‖)和一元逻辑非运算符(!)。通过这 3 种逻辑运算符的组合,可以用简单的条件构造出复杂的条件,以实现强大的逻辑测试功能。3 种逻辑运算中,逻辑非(!)的优先级最高,逻辑与(&&)次之,逻辑或(‖)最低。在结合性质上,逻辑非(!)是右结合,而逻辑与(&&)和逻辑或(‖)为左结合。逻辑表达式的语法描述如下：

```
<逻辑表达式>→<布尔型常量>|
              <布尔型变量>|
              <关系表达式>|
              <操作数>&&<操作数>|
              <操作数>||<操作数>|
              !<逻辑表达式>|
              (<逻辑表达式>)
```

可见，关系表达式是一种基本的逻辑表达式。下面是一组逻辑表达式的例子：

```
(grade>=60) && (grade<=70)
```

表示测试 grade 是否在 60 和 70 之间（包括 60 和 70）。

```
(year % 400==0) || (year % 100 !=0) && (year % 4==0)
```

表示测试 year 是 400 的倍数或者是 4 的倍数但不是 100 的倍数，这实际上是判断 year 的值是否为闰年数。

在计算逻辑表达式时，C++ 遵循一种所谓的“短路计算法”，即如果按照逻辑运算符的优先顺序和结合性质，只要计算部分表达式就可以确定整个表达式的结果，就不再计算表达式的剩余部分。以上面的表达式为例，计算（grade>=60）&&（grade<=70）时，如果 grade 是小于 60 的，&& 左边的子表达式（grade>=60）的结果为 false，此时无须计算右边的子表达式（grade<=70），就可以知道整个表达式为 false。对于 A&&B 来说，只有当计算了 A 得到的结果为 true 时，才需要计算 B 的值，以确定整个表达式的值。与此类似，对于逻辑或运算，只有当左操作数的结果为 false 时，才需要计算右操作数的值，该值就是整个表达式的结果。

2.5.4 位运算

位运算是 C 语言的重要特色之一，C++ 保留了这一特色。位运算允许在二进制位级别上对数据进行处理。位运算的语法如下：

```
<位运算表达式>→~<操作数>|
                <操作数>&<操作数>|
                <操作数>|<操作数>|
                <操作数>^<操作数>|
                <操作数><<<操作数>|
                <操作数>>><操作数>
```

各种位运算符说明如表 2-8 所示。

这些运算符中，除了～是一元运算符以外，其余的都是二元运算符，操作数都只能是整型或字符型数据，不能为浮点型数据，结果为整型或字符型。

表 2-8 位运算符说明

<table>
<tr><th>运算符</th><th>名　称</th><th>示　例</th><th>说　明</th></tr>
<tr><td>&</td><td>按位与</td><td>a&b</td><td>a 和 b 的每一位作与运算</td></tr>
<tr><td>|</td><td>按位或</td><td>a|b</td><td>a 和 b 的每一位作或运算</td></tr>
<tr><td>^</td><td>按位异或</td><td>a^b</td><td>a 和 b 的每一位作异或运算</td></tr>
<tr><td>~</td><td>按位取反</td><td>~a</td><td>将 a 的每一位取反</td></tr>
<tr><td><<</td><td>向左移位</td><td>a<<b</td><td>将 a 的每一位向左移 b 位</td></tr>
<tr><td>>></td><td>向右移位</td><td>a>>b</td><td>将 a 的每一位向右移 b 位</td></tr>
</table>

1. 按位与 &

& 运算结果的每一位是两个操作数二进制表示的对应位进行与运算的结果,即如果两个操作数的对应位都为 1,则结果的对应位也为 1,否则为 0。

例如,3 & 14 结果为 2。具体计算过程如下。

3 的二进制表示:00000011。

14 的二进制表示:00001110。

3 & 14 结果的二进制表示:00000010。

2. 按位或|

|运算结果的每一位是两个操作数二进制表示的对应位进行或运算的结果,即如果两个操作数的对应位都为 0,则结果的对应位也为 0,否则为 1。

例如,3|14 结果为 15。具体计算过程如下。

3 的二进制表示:00000011。

14 的二进制表示:00001110。

3|14 结果的二进制表示:00001111。

3. 按位异或^

^运算结果的每一位是两个操作数二进制表示的对应位进行异或运算的结果,即如果两个操作数的对应位不相同,则结果的对应位为 1,否则为 0。

例如,3^14 结果为 13。具体计算过程如下。

3 的二进制表示:00000011。

14 的二进制表示:00001110。

3^14 结果的二进制表示:00001101。

4. 按位取反~

~运算结果的每一位是操作数二进制表示的对应位进行取反运算的结果,即如果操作数的对应位为 0,则结果的对应位为 1,否则为 0。

例如,14 的二进制表示:00001110。

~14 结果的二进制表示:11110001。

5. 向左移位<<

<<运算将左操作数的二进制表示向左移位,移动的位数就是右操作数的值,右端移出的空位填充 0,移位后的左操作数的值即为运算的结果。

例如,3 <<5 的具体计算过程如下。

3 的二进制表示:00000011。

3 << 5 的结果:01100000。

6. 向右移位>>

>>运算将左操作数的二进制表示向右移位,移动的位数就是右操作数的值,移位后的左操作数的值即为运算的结果。左端移出的空位填充方式取决于左操作数的类型和具体的值:如果左操作数是无符号类型,或者是有符号类型但其值非负(最高位为 0),那么高位填充 0;如果左操作数是有符号类型,并且为负数(最高位为 1),高位填充的值取决于所用的计算机系统,有的 C++ 实现填充 0,有的填充 1。

例如,−7>>5 的具体计算过程如下。

−7 的二进制补码表示:11111001。

−7>>5 的结果:00000111 (填充 0)。

−7>>5 的结果:11111111 (填充 1)。

对于 &、| 和^等二元位运算,如果参与位运算的操作数类型不同,系统将二者按右端对齐,并根据操作数的类型和值进行填充,使得两个操作数位数完全一样。例如,如果 a 是 short int,占 2 个字节,b 是 int,占 4 个字节,则系统将按照 a 的符号位扩展成 4 个字节,即如果 a 非负,则用 0 扩展,如果 a 为负数,则用 1 扩展。如果需扩展的操作数是无符号类型,则总是用 0 扩展。

位运算符的优先顺序如下:按位取反~的优先级最高,向左移位<<及向右移位>>次之,然后依次是 &、^和 |。

位运算为程序员提供了非常灵活的、直接对二进制位进行处理的手段。通过各种位运算的合理组合,可以实现很多有趣的功能,见表 2-9。

表 2-9 位运算的组合示例

功　能	表　达　式	说　明
逐位清零	a&0	将 a 的每一位清零
取指定的二进制位	a&0X00FF	取 a 的低位字节
设置指定位	a|0X00FF	将 a 的低位字节的每一位设置为 1
指定位翻转	a^0X00FF	将 a 的低位字节的每一位置翻转,0 变为 1,1 变为 0
乘以 2 的幂	a<<n	a 乘以 2^n
除以 2 的幂	a>>n	a 除以 2^n

2.5.5 逗号运算

C++ 提供了一种特殊的运算符——逗号运算符。该运算符将两个表达式连接起来构成逗号表达式:

```
<逗号表达式>→<表达式>,<表达式>, …
```

逗号运算符是一个二元运算符，具有左结合性质，其优先级比前面介绍的运算都要低。其计算过程是：先计算逗号左边的表达式，再计算逗号右边的表达式，并且将右边的表达式的计算结果作为整个表达式的结果。例如：

12+4，3*5　　　　表达式的结果为15。

12+4，3*5，4-1　表达式的结果为3。

大多数情况，使用逗号表达式的目的是为了顺序计算多个表达式的值，而并非一定要获得逗号运算的结果。第4章将会看到逗号表达式常用于循环语句中。

2.5.6 赋值运算

赋值运算实现了对变量的赋值，即为已声明的变量赋予一个特定值。需要特别指出的是，C++的赋值运算符除了对变量进行赋值以外，作为一种运算符，还具有运算的结果，这是C++与很多其他程序设计语言不同的地方。

赋值运算符是二元运算符，也就是说，它具有两个操作数。赋值运算符和这两个操作数一起构成了赋值表达式，其中左边的操作数必须具有左值的表达式，其语法如下：

```
<赋值表达式>→<变量><赋值运算符><表达式>
<赋值运算符>→= | *= | /= | %= | += | -= |
               >>= | <<= | &= | ^= | |=
```

上述所有赋值运算符具有相同的优先级，其优先级比逗号运算符高，但比其他运算符的优先级低。赋值运算符具有右结合性质，即相邻的赋值运算符从右算到左。注意，赋值运算并不仅限于给变量定值，还可以用于给数组元素、指针等的定值。

=是简单赋值运算，其功能是：先将赋值运算符右边表达式的值计算出来，将该值赋给赋值运算符左边的变量，并将该值作为赋值运算的结果。因此，赋值表达式 a=3+5 表示将3与5的和8赋给变量a，整个赋值表达式的计算结果也是8。

同理，若a和b均为整型变量，且a的值为5，赋值表达式 b=a++计算完成后，a的值为6，而b的值为5。

对于其他如*=、/=、%=、+=、-=、>>=、<<=、&=、^=和|=等复合赋值运算符的使用(注意，这些运算符的两个字符之间不能留空格)，都可以概括为下面的模式：

```
变量 操作符=表达式
```

它们的功能相当于：

```
变量 = 变量 操作符 表达式
```

具体操作是：先计算赋值运算符右边的表达式的值，将作为左操作数的变量的当前值与该值进行操作符代表的运算，运算的结果复制给变量并作为整个表达式的结果。例如，假设a为整型变量，当前值为10，那么 a += 18/3 的计算过程为：先计算18除以3的整商，得到整数6，将变量a的当前值10与6求和，得到16，再将16赋给变量a，整个表达式的结果也是16。

需要指出的是，如果赋值运算中，赋值运算符两边的变量和表达式的类型不同，C++ 先隐含地将表达式的结果转换成变量的类型，再将转换的结果赋给变量。当然也可以显式地转换表达式的值的类型。因此，赋值运算的结果的类型总是与赋值运算符左边的变量的类型一致。例如，如果 a 为浮点变量，b 和 c 为整型变量，那么 a＝b＋c 计算过程为：先对整型变量 b 与 c 的值进行整数求和，然后将结果转变成浮点类型，再赋给浮点型的变量 a，同时，该浮点值也是整个表达式的值。下面的表达式具有同样的效果。

```
a=(float)(b+c)
```

注意：*类型转换也可以看成是一种运算符，其优先级比乘、除运算符要高。*

另外，C++ 允许在声明变量的时候通过等号对变量进行初始化，给变量定初始值。这个初始值可以是一个表达式，但是要求这个表达式的值在编译时是可计算的，即要求该表达式的值是一个常量。下面是一些合法的带初始化的变量声明。

```
int i=2;
double x=i+1.0;
double a=i*x;
```

经过声明和初始化后，变量 i 的值为 2，x 的值为 3.0，a 的值为 6.0。

2.5.7 特殊运算符

C++ 还有一些特殊运算符，本节先学习条件运算符，在后面的章节中将介绍其他特殊运算符。

条件运算符是一个三目运算符。该运算的一般形式为：

<表达式 1>?<表达式 2>:<表达式 3>;

条件运算符的含义是：先计算<表达式 1>的值，如果为 true(非 0)，则计算<表达式 2>的值，并把该值作为整个表达式的值；如果<表达式 1>的值为 false(为 0)，则直接计算<表达式 3>的值，并把它作为整个表达式的值。

例如，假设整型变量 x 的值为 90，y 为字符型变量，则表达式

```
y=x>70?'P':'F'
```

计算过程是：计算条件表达式 x>70 的值，结果为 true，所以直接将'P'作为赋值运算符右边表达式的值，并赋给 y，该值也是整个表达式的计算结果。

可见，利用条件运算符，可以根据条件完成不同的计算。

习 题 2

2.1 列举 C++ 语言中的各种基本数据类型，说明其特点以及定义方式。

2.2 C++ 中有哪些种类的运算符？各有什么功能？

2.3 赋值运算符有哪些？有什么特点？

2.4 什么是表达式？C++ 中有哪些表达式？如何计算表达式的值及确定其类型？

2.5 什么是一元、二元、三元运算符?它们在使用时应注意些什么?

2.6 什么是变量?变量的三要素是什么?

2.7 什么是常量?C++中常用的常量有哪些类型?

2.8 字符常量和字符串常量有何不同?举例说明。

2.9 什么是隐式类型转换?其规则是什么?什么是强制类型转换?

2.10 简述标识符定义。指出下列用户自己定义的标识符中哪些是合法的?哪些是非法的?如果是非法的,为什么?

xy Book 3ab x_2 switch integer

page-1 _name MyDesk #NO y.5 char

2.11 下列常量中,哪些是合法的?

78 063 c56 0x98 '\07' "\"b"" "abc\n" "\qaabs"

2.12 下列的常量是什么类型?

65538 123u 1.2 1.5E4F 7L 9uL

2.13 将下列数学表达式改写为表达式。

(1) $\frac{\cos x}{a+b+c}$ (2) $\sqrt{(x+y)^2+(x-y)^2}$

(3) $\sin\alpha/\cos\beta+\tan\delta$ (4) $\frac{|x-y|}{z+x}$

2.14 假设 a=4,b=10,c=2,计算下面算术表达式的值。

(1) a+b*c/(a+c)%3/a

(2) (float)(a+c)/3+(b-a)%a

2.15 假设 x=3,y=10,z=12,判断下面关系表达式或者逻辑表达式的真假。

(1) x-y>y-z (2) x<=y&&(x>0)||(y>0)

(3) !(x-y)>0||(y-z>0) (4) x+z==y||z-x<0

2.16 执行下列语句后,变量 a、b、c 的值各为多少?

```
int  a,  b,  c;
a=20;
b=++a;
c=a++;
```

2.17 若 x=3,y=2,z=1,下列各式的结果是什么?

(1) x|y&z (2) x|y-z (3) x^y&-z (4) x<<=2 (5) y<<2

2.18 若 x=1,y=-1,下列各式的结果是什么?

(1) !x|x (2) ~x|x (3) x^x (4) x<<=2 (5) y<<2

2.19 设计一个程序,测试你的计算机是如何处理下面的移位运算的:

(1) 如果向左移位运算<<和向右移位运算>>的右操作数是负数,结果是什么?

(2) 如果向右移位运算>>的左操作数是有符号类型,并且为负数(最高位为1),高位填充的值是0还是1?

2.20 写出下面表达式运算后 a 的值,设原来 a 的初始值为12。

(1) a+=a (2) a-=2 (3) a*=2+3 (4) a/=a+a

(5) a%=(n%=2),n 的值等于5 (6) a+=a-=a*=a

2.21 设计一个程序,从键盘输入一个小写字母,将它转换成大写字母。

2.22 设计一个程序,从键盘输入一个圆的半径,求其周长和面积。

第 3 章 输入和输出

【学习内容】

本章将介绍 C++ 的输入输出机制。主要内容包括：

◆ 标准输入输出函数。

◆ 格式化输入输出函数。

◆ 通过流进行输入输出。

【学习目标】

◆ 掌握各种输入输出函数。

◆ 掌握通过流进行输入输出的方法。

程序的输入输出(input/output,I/O)为用户提供与计算机进行交互的功能。用户可以通过程序的输入功能将执行意图和需要处理的数据传递给计算机,而计算机又通过程序的输出功能将对数据的处理结果告知用户。I/O 是程序的基本组成部分,具有重要的意义。

3.1 C++ 的输入和输出

C++ 保留了 C 的 I/O 功能,允许通过标准的 I/O 函数完成相应的操作。在此基础上,还增加了一些扩展的 I/O 功能,这些扩展功能都是面向对象的,可以通过 I/O 流对象来实现相应的功能。本章介绍两种 I/O 方式：一种是保留自 C 的 I/O 方式;另一种是面向对象的 I/O 方式。

相对于 C 的 I/O 而言,C++ 的 I/O 是类型安全的。各种 I/O 操作都是与类型相关的。也就是说,可以规定某 I/O 操作处理的数据的类型,当需要对数据进行该操作时,编译程序会检查数据的类型是否是规定的数据类型,如果不符合,就会产生编译错误,从而保证了处理的数据类型与操作匹配。在这一点上,C 存在着缺陷,可能会导致一些运行时的错误。此外,C++ 在 I/O 方面提供了很好的可扩展性,程序员可以设计自定义的 I/O 操作,而这些自定义的 I/O 与 C++ 自身的预定义的 I/O 在使用时可以采取相同的方式,为软件设计提供了非常好的可扩展性,有利于提高软件的重用性。

与 C 一样,C++ 的 I/O 是以字节流的形式实现的,所谓“流”实际上就是一个字节序列。在输入操作中,字节从输入设备流向内存;在输出操作中,字节从内存流向输出设备。有 3

种预定义的标准流：标准输入流，一般代表键盘，用于向程序输入数据；标准输出流，一般代表显示器，用于显示程序的执行结果；标准错误流，一般也是显示器，用于系统报告错误信息。

有关输入输出的操作并没有在C++语言中定义，但它包含在C++的实现中，并作为C++的一个标准类库提供。

软件工程提示：尽管C++兼容了C的I/O，程序设计时还是应当尽可能采用C++的I/O方式，该方式功能更强，使用更方便，软件可扩展性和可重用性更高。

3.2　标准输入输出函数

C++继承了C语言的输入输出函数库。下面先介绍从C保留下来的、最简单的标准输入输出函数：getchar函数和putchar函数，它们只能用来输入和输出一个字符。这两个函数的定义都在stdio.h文件中。

3.2.1　字符输入函数getchar

getchar函数的原型为：

```
int getchar(void);
```

getchar函数没有参数，它从标准输入设备(一般是键盘)读入一个字符，读入的字符以整数形式返回，而且将该字符回显在显示器屏幕上。该函数原型包含在stdio.h文件中。与C语言一样，文件stdio.h包括与输入输出相关的库函数的定义和使用信息。如果要使用其中的数据和函数，一定要用预处理包含指令#include将该文件包含在应用程序中。

3.2.2　字符输出函数putchar

putchar函数的原型为：

```
int putchar(int c);
```

putchar函数向标准输出设备(一般是显示器屏幕)输出一个字符，它的参数c是要输出的字符变量或常量。这个函数执行成功时，返回要输出的字符变量c；否则，返回EOF。putchar函数的原型也包含在stdio.h文件中。

下面通过例子来进一步理解字符I/O函数的使用。

【例3-1】　字符输入输出函数的应用。

```
//ex3_1.cpp
#include<stdio.h>
int main()
{
    char a='a', b='b', c;
    c=getchar();
```

```
    putchar(a);
    putchar(b);
    putchar('c');
    putchar(c);
    putchar('\"');
    putchar(0101);
    putchar('\n');

    return 0;
}
```

程序运行时,从键盘输入一个字符 B,系统同时在屏幕上显示 B,然后继续显示后续输出语句的输出信息:

```
B
abcB"A
```

注意:程序中的 0101 是字母 A 的 ASCII 代码的八进制表示;'\"'代表双引号符;'\n'代表换行符,系统输出该符号时将下一个输出符号的位置调整到下一行的开始。

3.3 格式化输入输出函数

为了提高程序的用户友好性,通常需要用户将数据按照一定格式输入,程序的执行结果也需要按照一定的格式组织输出,以获得有序、美观的效果。为此,C++ 提供了格式化输入输出手段。常用的格式化输入输出函数有两个:printf 和 scanf。printf 函数按照指定的格式向屏幕输出数据,scanf 函数按照一定的格式从键盘输入数据。

3.3.1 格式化输入函数 scanf

1. 格式化输入函数的原型

scanf 是常用的格式化输入函数,其原型在 stdio.h 文件中说明。scanf 函数按照一定的格式从键盘输入数据,其功能主要有:

- 输入各种类型的数据,并存入相应的参数中。
- 读取输入流中的指定的字符。
- 跳过输入流中的指定的字符。

scanf 的使用格式如下:

```
scanf (<格式控制串>,<参数列表>);
```

其中,<格式控制串>一般是一个字符串常量,也可以是字符串指针,描述了输入数据应该遵循的格式。格式控制串可以包含 3 种类型的字符:格式指示符、空白字符(空格符、制表符和回车符)和非空白字符(又称普通字符)。

<参数列表>是存放输入数据地址的列表,输入数据的个数是可变的,参数之间用“,”分隔。实际上输入参数列表是一个地址列表,可以是变量的首地址,也可以是字符数组名或指针变量(将在第 6、7 章介绍)。变量首地址的表示方法为:

```
&变量名
```

其中,& 是地址运算符,其运算结果是后面的变量的首地址。

格式指示符的数量、顺序与输入参数对应。scanf 函数执行时,根据格式控制串中的格式指示符,读取或跳过输入流中的数据,按照格式指示符描述的类型信息,将输入数据转换成相应的类型并存入对应的输入参数。

注意:scanf 函数有返回值,如果输入成功,该函数返回转换成功的参数的数目;否则,返回文件结束标志符 EOF 表示错误。

例如,对于下面的函数:

```
scanf("%d %f %d",&num1, &num2, &num3);
```

其中,num1 和 num3 为整型变量,num2 为浮点类型的变量,%d 和%f 均为格式指示符,分别表示读取的数据为整型和浮点型。如果输入为

```
12 34.5 678
```

则该函数执行的效果是:系统读取整数 12 并赋值给 num1,读取浮点数 34.5 并赋值给 num2,读取整数 678 并赋值给 num3。

2. scanf 函数的格式指示符

对于 scanf 函数,格式控制串中的空白字符作为相邻输入数据的默认分隔符,非空白字符在输入有效数据时,必须原样一起输入。格式控制串中格式指示符的一般形式为:

```
%*<宽度><转换说明符>
```

各部分说明如下。

1) 格式控制字符标志%

%是格式控制字符的标志,它是格式指示符的前导符。

2) 赋值抑制符 *

赋值抑制符 * 是任选项,表示该格式指示符对应的数据读入后,不赋给相应的变量,该变量通过下一个格式指示符输入数据。例如,对于输入函数:

```
scanf("%d%*d%d%d",&num1,&num2,&num3);
```

如果 num1、num2 和 num3 均为整型变量,假设输入为:

```
12 34 567 89
```

则系统将读取 12 并赋值给 num1;读取 34 但舍弃掉(* 阻止将该数据赋值给 num2);读取 567 并赋值给 num2,读取 89 并赋值给 num3。

3) 宽度

宽度也是任选项,为一整数 n,指明了该项输入数据所占列数为 n,即读取输入数据中相应的 n 位。赋值时,按参数所需要的位数赋给相应的变量,如果有多余的部分将被舍弃。例如,

```
scanf("%3c%3c",&ch1,&ch2);
```

如果 ch1 和 ch2 均为字符型变量，格式指示符%3c 指明读取数据的宽度为 3，读取数据的类型为字符型。假设输入为：

```
abcdefg
```

则系统将先读取 abc，并将其中的第一个字符 a 存入变量 ch1；然后继续读取 def，并将其中第一个字符 d 存入变量 ch2。

4) 转换说明符

转换说明符指明了读取数据的类型信息。%d、%f 和%c 分别表示读取的数据为整型、浮点型和字符型。表 3-1 给出了常用的转换说明符及其意义。

表 3-1 scanf 函数常用的转换说明符

类型字符	含　义
d	十进制整型量
o	八进制整型量
x	十六进制整型量
u	无符号十进制整型
i	整型
f	实型的小数形式
e	实型的指数形式
g	f 和 e 的较短形式
c	字符
s	字符串
l 或 h	放在任何整数转换说明符之前，用于输入 short 或 long 类型数据
l 或 L	放在任何浮点转换说明符之前，用于输入 double 或 long double 类型数据

3.3.2 格式化输出函数 printf

1. **格式化输出函数的原型**

printf 是常用的格式化输出函数，其原型也是在 stdio.h 文件中说明的。printf 函数按照一定的格式向标准输出设备(如显示器)输出数据，实现下面格式输出功能：

- 指定浮点值保留的小数位数。
- 浮点值小数点对齐。
- 输出数据的右对齐和左对齐。
- 将字符直接插入到输出数据中。
- 将浮点数按照指数形式输出。
- 将整数按照八进制或十六进制形式输出。

- 按指定的域宽和精度输出数据。

与 scanf 类似，printf 的使用格式如下：

```
printf (<格式控制串>,<参数列表>);
```

<格式控制串>一般是一个字符串常量，也可以是字符串指针，以描述输出数据的格式。格式控制串可以包含3种类型的字符：格式指示符、转义字符、普通字符（除格式指示符和转义字符外的其他字符）。格式指示符的作用是将参数列表中相应的输出数据转换为指定的格式输出，转义字符按照其含义输出相应的特殊符号，而格式控制字符则被参数列表中相应的输出数据所替换。

<参数列表>是存放输出数据的表达式列表，输出数据的个数可变，参数之间用“,”分隔。格式指示符的数量、顺序与输出参数相对应。printf 函数执行时，根据格式控制串中的格式指示符、转义字符和普通字符等，从左到右依次进行处理：遇到普通字符，则按照原样输出；遇到转义字符，则按照其含义输出相应的符号；遇到格式控制字符，则先计算相应的输出参数表达式，将该参数的计算结果按照格式指示符指定的方式输出。

继续考虑上节的例子，对于下面的程序：

```
scanf("%d%f%d",&num1, &num2, &num3);
printf("Three number: num1=%d, num2=%f, num3=%d\n sum=%f",num1, num2, num3, num1+num2+num3);
```

其中，num1 和 num2 为整型变量，num3 为浮点类型的变量，如果输入为：

```
12 34.5 678
```

则 scanf 函数执行完后，num1 的值为 12，num2 的值为 34.5，num3 的值为 678；接着执行 printf 函数，将 3 个变量的最新值进行格式化输出：

```
Three number: num1=12, num2=34.500000, num3=678
sum=724.500000
```

注意：printf 函数的格式控制串中的格式指示符%d 和%f 分别表示参数的值按照整型和浮点型形式输出，其中浮点型数据缺省则保留 6 位小数位。

2. 格式指示符

对于 printf 函数，不同类型的数据，要使用不同的格式指示符，格式控制串中的格式指示符的一般形式为：

```
%<标志><域宽><.精度><转换说明符>
```

其中，<标志>、<域宽>和<精度>都是任选项，可以不出现。

各部分说明如下。

1）标志

标志紧跟在%的后边，可以增强格式化输出的能力，程序员可以使用表 3-2 所列的 5 种标志以实现不同的格式控制功能。

表 3-2　printf 函数常用的标志

标志	含　　义
-	输出在域宽内左对齐
+	在正数值之前显示一个加号，在负数值之前显示一个减号
空格	在正数值之前显示一个空格
#	与八进制转换说明符 o 一起使用时，在输出值之前加 0；与十六进制转换说明符 x 或 X 一起使用时，在输出值之前加 0x 或 0X
0	用 0 填充域宽

注意：多个标志可以联合使用。

2）域宽

域宽为一整型表达式，指明了数据打印的宽度。如果数据实际长度小于域宽，则数据输出右对齐，即数据仍然按照域宽规定输出，在数据左边用空格补齐；如果数据实际长度大于域宽，那么系统将自动突破域宽的限制，按照数据的实际长度进行输出。注意，负号要占据一个字符位置。如果没有指明域宽，系统按照数据的实际长度输出。

3）精度

精度也是一个整型表达式，对于不同类型的数据，精度的含义也不一样。对于整数，精度表示至少要打印的数字个数，如果数据实际长度小于精度，则左边补齐 0，使得数据长度等于精度；如果数据实际长度大于精度，则自动突破精度限制，按照数据的实际长度输出。缺省情况下整数的精度为 1。对于浮点数，如果转换说明符为 e、E 和 f，那么精度表示小数点后的有效位数，如果数据小数部分的实际长度小于精度，则在右边补齐 0，使得小数部分长度等于精度；如果数据小数部分的实际长度大于精度，则按照精度对数据进行舍入输出。如果浮点数的转换说明符为 g 和 G，那么精度表示打印数据的最大的长度。对于字符串数据，精度表示字符串输出的最大长度，如果输出字符串的实际长度小于精度，则按照字符串的实际长度输出；如果字符串的实际长度大于精度，则按照精度截取输出字符串开头的 n 个字符（假设精度为 n）输出。

4）转换说明符

转换说明符指明了输出数据的类型信息。%d、%f 和%c 分别表示读取的数据为整型、浮点型和字符型。表 3-3 给出了常用的转换说明符及其意义。

表 3-3　printf 函数常用的转换说明符

类型字符	含　　义
d	十进制整型量
o	八进制整型量
x	十六进制整型量
u	无符号十进制整型
i	整型
f	实型的小数形式

续表

类型字符	含　义
e	实型的指数形式
g	f 和 e 的较短形式
c	字符
s	字符串
%	输出%本身
l 或 h	放在任何整数转换说明符之前,用于输出 short 或 long 类型数据
L	放在任何浮点转换说明符之前,用于输出 long double 类型数据

下面通过几个示例来进一步理解格式化输入输出函数的功能。

3.3.3 格式化输入/输出函数应用举例

下面的几个程序显示了各种格式化输入和输出功能的应用,注意程序中的注释,比较程序的输出结果。

【例 3-2】 说明基本格式化输入和输出的功能。

```
//ex3_2.cpp: 基本格式化输入和输出
#include<stdio.h>
int main()
{
    int num1;
    float num2;
    char ch1;
    int na, nb, nc, nd, ne, nf, ng;
    double da, db, dc;
    //基本格式化输入和输出
    printf ("-------------Basic input and output:-------------\n");
                        //输出时无参数,仅输出字符串
    printf("Please input a character, a integer and a float : ");
                        //提示用户输入数据
    scanf("%c %d %f", &ch1,&num1, &num2 );
                        //依次读取字符、整数和浮点数分别存入 ch1、num1 和 num2
    printf("Input data are: ch1=%c, num1=%d, num2=%f\n", ch1, num1, num2);
                        //输出读取的用户数据
    //转换说明符的使用
    printf("\n-----------Scanf with conversion specifier:-----------\n");
    //整数
    printf( "Please enter seven integers: " );
    scanf( "%d%i%i%i%o%u%x", &na, &nb, &nc, &nd, &ne, &nf, &ng );          //读取各种形式的整数
    printf( "The input displayed as decimal integers is:\n" );
```

```
    printf( "%d%d%d%d%d%d%d\n", na, nb, nc, nd, ne, nf, ng );
                                              //以十进制形式输出读取的用户数据
    //浮点数
    printf( "\nEnter three floating-point numbers: " );
    scanf( "%lf%le%lg", &da, &db, &dc );      //读取各种形式的浮点数
    printf( "Here are the numbers entered in plain floating-point notation:\n" );
    printf( "%f\n%f\n%f\n", da, db, dc );     //输出读取的用户数据
    return 0;
}
```

程序执行的结果为：

```
-------------Basic input and output:-------------
Please input a character, a integer and a float : A 12 34.5
Input data are: ch1=A, num1=12, num2=34.500000
-------------Scanf with conversion specifier:-------------
Please enter seven integers:-70 -70 070 0x70 70 70 70
The input displayed as decimal integers is:
-70 -70 56 112 56 70 112
Enter three floating-point numbers: 1234.56 1234.56e-04 1234.56e+4
Here are the numbers entered in plain floating-point notation:
1234.560000
0.123456
12345600.000000
```

【例 3-3】 说明输入宽度、赋值抑制符 * 的使用。

```
//ex3_3.cpp: 输入宽度、赋值抑制符 * 的使用
#include<stdio.h>
int main()
{
    int num1;
    float num2;
    int num3;
    char ch1;
    char ch2;
    //输入宽度的使用
    printf("-------------Scanf with width:-------------\n");
    printf("Please enter a six characters string: ");      //提示用户输入数据
    scanf("%3c%3c",&ch1, &ch2);                            //依次读取字符存入 ch1 和 ch2
    printf("Input data are: ch1=%c, ch2=%c\n", ch1, ch2);  //输出读取的用户数据
    printf("\nPlease enter a six digits integer: " );      //提示用户输入数据
    scanf("%2d%d", &num1, &num3 );                //依次读取两个整数分别存入 num1 和 num3
    printf("Input data are: num1=%d, num3=%d\n", num1, num3);    //输出读取的用户数据
    //输入赋值抑制符 * 的使用
```

```
    printf ("\n-----------Scanf with suppresses assignment:-----------\n");
    printf("Please input 2 integers, a float and a integer: ");          //提示用户输入数据
    scanf("%d%*d%f%d",&num1, &num2, &num3);
              //依次读取整数存入 num1,跳过第二个整数,读取浮点数存入 num2,读取整数存入 num3
    printf("Input data are: num1=%d, num2=%f, num3=%d\n", num1, num2, num3);
                                                                          //输出读取的用户数据

    return 0;
}
```

程序执行的结果为:

```
-------------Scanf with width:-------------
Please enter a six characters string: abcdef
Input data are: ch1=a, ch2=d

Please enter a six digits integer: 123456
Input data are: num1=12, num3=3456

-------------Scanf with suppresses assignment:-------------
Please input 2 integers, a float and a integer: 123 45 6.78 90
Input data are: num1=123, num2=6.780000, num3=90
```

【例 3-4】 说明格式化输出功能。

```
//ex3_4.cpp: 格式化输出功能
#include<stdio.h>
int main()
{
    int num1;
    double num2;

    //格式化输出
    //转换说明符的使用
    printf ("-----------Print with conversion specifier:-----------\n");
    //整数
    printf( "Print integers:\n");
    printf( "%d\n", 455 );
    printf( "%i\n", 455 );
    printf( "%d\n",+455 );
    printf( "%d\n",-455 );
    printf( "%hd\n", 32000 );
    printf( "%ld\n", 2000000000 );
    printf( "%o\n", 455 );
    printf( "%u\n", 455 );
    printf( "%u\n",-455 );
    printf( "%x\n", 455 );
    printf( "%X\n", 455 );

    //浮点数
```

```
printf( "\nPrint floats:\n");
printf( "%e\n", 1234567.89 );
printf( "%e\n",+1234567.89 );
printf( "%e\n",-1234567.89 );
printf( "%E\n", 1234567.89 );
printf( "%f\n", 1234567.89 );
printf( "%g\n", 1234567.89 );
printf( "%G\n", 1234567.89 );
//字符
printf( "\nPrint character:\n");
printf( "%c\n", 'A' );
//宽度
printf("\nPrint with width:\n");
printf( "%4d\n", 1 );
printf( "%4d\n", 12 );
printf( "%4d\n", 123 );
printf( "%4d\n", 1234 );
printf( "%4d\n\n", 12345 );
printf( "%4d\n",-1 );
printf( "%4d\n",-12 );
printf( "%4d\n",-123 );
printf( "%4d\n",-1234 );
printf( "%4d\n",-12345 );
//精度
printf ("\n-------------Print with precisions:-------------\n");
num1=873;
num2=123.94536;
printf( "Using precision for integers\n" );
printf( "\t%.4d\n\t%.9d\n\n", num1, num1 );
printf( "Using precision for floating-point numbers\n" );
printf( "\t%.3f\n\t%.3e\n\t%.3g\n\n", num2, num2, num2);
printf( "Using precision for strings\n" );
printf( "\t%.11s\n", "Hello World!" );
//标志
printf ("\n-------------Print with flags:-------------\n");
printf( "%10s%10d%10c%10f\n", "hello", 7, 'a', 1.23 );
printf( "%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23 );
printf( "%d\n%d\n", 786,-786 );
printf( "%+d\n%+d\n", 786,-786 );
printf( "% d\n% d\n", 547,-547 );
printf( "%#o\n", 1427 );
printf( "%#x\n", 1427 );
printf( "%#X\n", 1427 );
printf( "\n%g\n", 1427.0 );
```

```
    printf( "%#g\n", 1427.0 );
    printf( "%+09d\n", 452 );
    printf( "%09d\n", 452 );

    return 0;
}
```

程序执行结果:

```
-------------Print with conversion specifier:-------------
Print integers:
455
455
455
-455
32000
2000000000
707
455
4294966841
1c7
1C7

Print floats:
1.234568e+006
1.234568e+006
-1.234568e+006
1.234568E+006
1234567.890000
1.23457e+006
1.23457E+006

Print character:
A

Print with width:
   1
  12
 123
1234
12345

  -1
 -12
-123
-1234
-12345
-------------Print with precisions:-------------
Using precision for integers
```

```
          0873
          000000873
Using precision for floating-point numbers
          123.945
          1.239e+002
          124
Using precision for strings
          Hello World
-------------Print with flags:-------------
     hello           7            a   1.230000
hello      7            a            1.230000
786
-786
+786
-786
547
-547
02623
0x593
0X593
1427
1427.00
+00000452
000000452
```

3.4 用流进行输入/输出

本章之初介绍了C++在C的I/O功能的基础上，还增加了一些扩展的I/O功能，这些扩展功能都是面向对象的，即可以通过I/O流对象来实现相应的功能。C++把进行数据传送操作的设备也抽象成对象，将“流”作为具有输入输出功能的外设(如键盘、显示器等)和程序之间通信的通道。当进行输入操作时，数据从键盘流向程序，输出时数据则从程序流向显示器。C++预定义了cout、cin、cerr和clog 4个与输入输出相关的对象。与这些对象相关联的流为程序提供了与特定的输入输出设备进行通信的通道：cout代表标准输出流，C++预定义cout代表标准输出设备，即显示器；cin代表标准输入流，C++预定义cin代表标准输入设备，即键盘；cerr和clog代表标准错误流对象，也预定义为显示器。这些对象与重载运算符<<和>>相结合，可以完成输入输出功能。

下面C++的I/O库指的是iostream类库。iostream类库提供了数百种I/O功能，iostream类库的接口部分包含在几个头文件中。头文件iostream.h包含了操作所有输入输出流所需要的基本信息，因此大多数C++程序都应该包含这个头文件。在iostream类库中，ostream和istream是两个非常重要的类，头文件iostream.h中定义的cin、cout、cerr和clog 4个对象，分别对应于标准输入流、标准输出流、非缓冲的标准错误流和经缓冲的标准

错误流,除了 cin 是 istream 类的对象外,其他 3 个都是 ostream 类的对象。该头文件提供了无格式 I/O 功能和格式化 I/O 功能。

3.4.1 通过 cout 流输出数据

流插入运算符(即重载的运算符<<)和 cout 结合在一起使用,可向显示器屏幕输出数据。一般形式为:

```
cout<<<表达式>;
```

它把<表达式>的值输出到屏幕上,该表达式可以是各种基本类型的常量、变量或者由它们组成的表达式。也可以是指针,以十六进制数输出(但如果是字符指针,则输出指针指向的字符串)。

输出时,程序根据表达式的类型和数值大小,采用不同的默认格式输出,大多数情况下可满足要求。若要输出多个数据,可以连续使用流插入运算符<<,具体形式如下:

```
cout<<<表达式 1> << <表达式 2> << <表达式 3>…;
```

为了增强输出的可读性,在输出多个数据时,可以通过插入空格符、制表符或其他提示信息将数据进行组织,以获得更好的效果。

下面的程序片断使用一条流插入语句显示字符串:

```
cout<<"Welcome to C++!\n";
```

输出结果是:

```
Welcome to C++!
```

也可以用多条流插入语句显示多个字符串信息:

```
cout<<"Welcome to ";
cout<<"C++!\n";
```

输出结果也是:

```
Welcome to C++!
```

还可以用流操纵算子 endl(行结束)实现转义序列\n(换行符)的功能。例如,程序段:

```
cout<<"Welcome to ";
cout<<endl;
cout<<"C++!";
```

与

```
cout<<"Welcome to \n";
cout<<"C++!";
```

具有相同的输出结果:

```
Welcome to
C++!
```

endl是一个流操纵算子，它发送一个换行符并刷新输出缓冲区(不管输出缓冲区是否已满都把输出缓冲区的内容立即输出)。也可以用下面的语句刷新输出缓冲区：

```
cout<< flush;
```

下面是流插入运算符在一条语句中连续使用，并且输出表达式计算结果的例子：

```
cout<< "123+ 456= "<< (123+ 456)<< endl;
```

输出结果是：

```
123+ 456= 579
```

3.4.2 通过cin流输入数据

流读取运算符(即重载的运算符>>)和cin结合在一起使用，可从键盘输入数据。一般形式为：

```
cin>><变量>;
```

其功能是从键盘读取一个数据并将其赋给<变量>。该变量一般是算术类型的。从键盘输入的数据的类型应和变量相一致。也可以连续使用>>，实现从键盘对多个变量输入数据，其形式为：

```
cin >> <变量 1> >> <变量 2> >> <变量 3>…;
```

这要求从键盘输入数据的个数、类型与变量列表相一致。各数据之间要有分隔符，分隔符可以是一个或多个空格符、制表符或回车符。

【例3-5】 流读取运算符>>和cin的使用。

```
//ex3_5.cpp: 流读取运算符>> 和 cin
#include< iostream.h>
int main()
{
    char c;
    int i;
    float x , y;
    cout<< "Enter: \n";
    cin>> i>> x>> y;
    c= i;
    cout<< "c= "<< c<< "\ti= "<< i;
    cout<< "\tx= "<< x<< "\ty= "<< y<< "\n";

    return 0;
}
```

程序运行时显示：

```
Enter:
```

这时从键盘输入一个整数和两个实数,中间用一个或多个空格键作分隔符。如输入:

```
65 2.3 3.5
```

最后屏幕显示:

```
c=A    i=65    x=2.3   y=3.5
```

程序中的'\t'和'\n'是转义符,'\n'和"\n"效果相同,表示换行符。字符变量和整型变量 i 的值都是 65,但输出的形式不同。使用标准流输入输出方便安全。本书大部分的例题中使用它们进行输入输出。

C++的 iostream 类库为格式化输入输出提供了灵活、方便、丰富的功能,更详细的介绍参见 16.4 节。

3.5 流操纵算子

除继承了 ANSI C 的格式化输入输出函数,C++针对流也提供了丰富的用于实现格式化输入输出的流操纵算子。通过这些流操纵算子,可以实现许多格式控制功能,如设置域宽、设置精度、设置和清除格式化标志、设置域填充字符、在输出流中插入空字符、跳过输入流中的空白字符等。流操纵算子的使用非常方便,直接用于流插入运算符之后,即可发生作用,直到遇到新的流操纵算子为止。表 3-4 列出了常用的流操纵算子及其功能描述。本节仅介绍设置整数基数、设置浮点数精度、设置域宽的流操纵算子。

表 3-4　常用的流操纵算子及其功能描述

流操纵算子	功 能 描 述
setbase(b)	以进制基数 b 为输出整数值
setprecision(n)	将浮点精度设置为 n
setw(n)	按照 n 个字符来读或者写
flush	刷新 ostream 缓冲区
ends	插入字符串结束符,然后刷新 ostream 缓冲区
endl	插入换行符,然后刷新 ostream 缓冲区
ws	跳过空白字符
setfill(ch)	用 ch 填充空白字符

3.5.1 设置整数基数的流操纵算子

printf 函数支持将整数按照十进制、八进制和十六进制等多种形式输出。标准输出流 cout 的流操纵算子 oct 可将整数输出形式设置为八进制(基数为 8)、流操纵算子 hex 可将整数输出形式设置为十六进制(基数为 16)、流操纵算子 dec 可恢复成十进制基数。默认情况下,cout 输出整数采用十进制形式。

此外，也可以用流操纵算子 setbase 来灵活地改变基数。该算子可带有一个整数参数 10、8 或 16，分别等价于 dec、oct 和 hex。使用 setbase 或其他任何参数化的流操纵算子都必须在程序中包含头文件 iomanip. h。一旦设置流输出整数的基数，流的基数就保持不变，直到遇到新的设置整数基数的算子。

【例 3-6】 设置整数基数的流操纵算子的使用。

```
//ex3_6.cpp: 设置整数基数的流操纵算子的使用
#include<iostream.h>
#include<iomanip.h>
int main()
{
   int n;
   cout<<"Enter a decimal number: ";
   cin>>n;
   cout<<n<<" in hexadecimal is: "
        <<hex<<n<<endl
        <<dec<<n<<" in octal is: "
        <<oct<<n<<endl;
   return 0;
}
```

程序执行结果：

```
Enter a decimal number: 48
48 in hexadecimal is: 30
48 in octal is: 60
```

3.5.2 设置浮点数精度的流操纵算子

在 C++ 中也可以控制浮点数输出的精度，流操纵算子 setprecision 或函数 precision 都可控制浮点数小数的有效数字位数。一旦设置了精度，该精度对 cout 中其后的所有的输出操作都有效，直到遇到下一个流操纵算子重新设置精度为止。以无参数的方式调用函数 precision，可以返回当前设置的精度。

【例 3-7】 设置浮点数精度的流操纵算子的使用。

```
//ex3_7.cpp: 设置浮点数精度的流操纵算子的使用
#include<iostream.h>
#include<iomanip.h>
#include<math.h>
int main()
{
   double log2=log(2.0);            //在头文件<math.h>中定义
   int places;
   cout<<"log(2) with precisions 0-9.\n"
```

```
            << "Precision set by the "
            << "precision member function:"<< endl;
    //使用 precision算子
    for ( places= 0; places< = 9; places+ + ) {
        cout.precision( places );
        cout<< log2<< '\n';
    }
    cout<< "\nPrecision set by the "
            << "setprecision manipulator:\n";
    //使用 setprecision算子
    for ( places= 0; places< = 9; places+ + )
        cout<< setprecision( places )<< log2<< '\n';
    return 0;
}
```

程序执行结果:

```
log(2) with precisions 0- 9.
Precision set by the precision member function:
0.7
0.7
0.69
0.693
0.6931
0.69315
0.693147
0.6931472
0.69314718
0.693147181
Precision set by the setprecision manipulator:
0.7
0.7
0.69
0.693
0.6931
0.69315
0.693147
0.6931472
0.69314718
0.693147181
```

3.5.3 设置域宽的流操纵算子

函数 width 可以实现对当前域宽(即输入输出的字符数)的设置,该函数同时返回原来

(上一次)设置的域宽。如果输出的数据所需的宽度比设置的域宽小,则空位用填充字符(默认为空格)填充。如果被显示的数据所需的宽度比设置的域宽大,则系统会自动突破宽度限制,输出所有位。流操纵算子 setw 也可以用来设置域宽。

【例 3-8】 设置域宽的流操纵算子的使用。

```
//ex3_8.cpp:设置域宽的流操纵算子的使用
#include<iostream.h>
#define WIDTH 5

int main()
{
   int w=4;
   char string[WIDTH+1];

   cout<<"Enter a sentence:\n";
   cin.width(WIDTH);

   while (cin>>string) {
       cout.width(w++);
       cout<<string<<endl;
       cin.width(WIDTH);
   }

   return 0;
}
```

程序执行结果:

```
Enter a sentence:
This is a demo sentence.
This
   is
     a
   demo
    sent
     ence
         .
```

C++的I/O库还提供了许多其他丰富的输入输出功能,读者可以参看相关的文献资料,并在实际应用中不断地学习和实践。

习　题　3

3.1　在标准输入输出中 cout、cerr、clog、cin 的作用是什么?有何区别?

3.2　哪个 I/O 操作符可在输出浮点数据时减少打印位数?

3.3　在用 I/O 操作符时,必须包括什么头文件?

3.4　如何理解流的概念?流的插入和提取是指什么?与其相一致的对象分别是什么?

3.5　用一条 C++语句实现下列要求:

(1) 以左对齐方式输出整数40000,域宽为15。
(2) 把一个字符串读入字符型数组变量state中。
(3) 打印有符号数200和无符号数200。
(4) 将十进制整数100以0x开头十六进制格式输出。
(5) 用前导0格式打印1.234,域宽为9。

3.6 写出下列程序的运行结果:

```
#include<iostream.h>
void main(void)
{   int n=4;
    while (--n)
      cout<<n<<'\t';
    cout<<endl;
}
```

3.7 编写一个程序,测试十进制、八进制、十六进制格式整数值的输入,并分别按照3种不同的基数输出,测试数据为10、010、0x10。
3.8 编写一个程序,分别用不同的域宽打印出整数12345和浮点数1.2345。当域宽小于数值的实际需要的域宽时,会发生什么情况?
3.9 编写一个程序,将100.453627取整到最近似的个位、十分位、百分位、千分位和万分位,打印出结果。
3.10 编写一个程序,利用流操纵算子输出下面数据:

```
12345678901234567890
432*******
*******432
0x11
45456.67
4.5457E+04
```

3.11 设计一个程序,判断从键盘输入的整数的正负性和奇偶性。
3.12 编写一程序,记录5个班的学生成绩,每个班有10个学生。要求用cin函数输入数据,然后按行列格式打印该表。
3.13 假定你是一大学教师,需要给出10个学生的平均分数。编写一个程序,提示输入10个不同的成绩,然后将平均值显示在屏幕上。
3.14 编写一个测试儿童算术的程序。要求输入两个数,然后算出第一个数加第二个数的和。当他准备好后(在告诉他怎样做以后等着他按Enter键)打印结果,以便他核对答案。
3.15 编写一个程序,将华氏温度0～212度转换为浮点型摄氏温度,浮点数精度为3。转换公式如下:

```
celslus=5.0/9.0*(Fahrenheit-32)
```

输出两个右对齐列,摄氏温度前面加上正负号。
3.16 从键盘输入一个3位数abc,从左到右用a、b、c表示各位的数字,现要求依次输出从右到左的各位数字,即输出另一个3位数cba,如输入123则输出321,试设计程序(算法提示:a=n/100,b=(n−a*100)/10,c=(n−a*100)%10,m=c*100+b*10+a)。

第 4 章 控制结构

【学习内容】

本章介绍 C++ 的基本控制结构,学习结构化程序设计方法。主要内容包括:

◆ 表达式语句。

◆ 顺序结构:复合语句。

◆ 选择结构:if 语句、if-else 语句和 switch 语句。

◆ 循环结构:while 语句、do-while 语句和 for 语句。

◆ 控制转移语句:continue 语句、break 语句和 goto 语句。

【学习目标】

◆ 熟练使用控制结构进行结构化程序设计。

◆ 掌握自顶向下、逐步求精的结构化程序设计方法。

程序从本质上说是描述一定数据的处理过程。对数据的加工和处理往往是复杂的。前面看到,通过计算表达式可以实现对数据的简单处理,如果要对数据进行更为复杂的处理,还需要利用程序设计语言提供的各种控制机制。在 C++ 中,这些控制机制都具有良好的结构性,通过组合可以实现强大的计算功能,满足复杂的需求。

4.1 程序的语句和基本控制结构

程序是由若干条语句组成的。不同程序语言都有不同形式和功能的各种语句。从功能上说,语句可以分为执行性语句和说明性语句两大类。说明性语句旨在声明各种不同数据类型的变量或运算,而执行性语句旨在描述程序的动作,即对数据如何进行操作。

构成一段程序的若干条语句可以按照顺序一条一条地执行,这种顺序结构是简洁的。但在现实世界中,很多问题的解决都难以严格按照顺序进行,不可避免地遇到需要进行选择、不断循环反复的情况。这时,控制的顺序会发生转移,而非从前向后逐一执行。程序的执行也是如此。因此,程序中除了顺序结构外,通常还有选择结构、循环结构和转移机制。

从结构化程序设计观点看,所有程序都可只用顺序结构、选择结构和循环结构 3 种控制结构实现,Bohm 和 Jacopini 的研究证明了这一点。C++ 在默认情况下采取顺序结构,除非特别指明,计算机总是按语句顺序一条一条地执行。为使程序更清晰、更易调试与修改,并且不容易出错,结构化编程要尽量少用或不用 goto 等跳转语句。

C++ 为了支持这些控制结构,提供了丰富、灵活的语句。C++ 除了声明语句外,还提供了 5 种形式的执行性语句:表达式语句、空语句、函数调用语句、复合语句(又称块)、控制语句等。

C++ 提供了 3 种选择结构,即 if 选择结构、if-else 选择结构和 switch 选择结构。if 选择结构又称为单分支选择结构,选择或跳过一个操作;if-else 选择结构也称为双分支选择结构,在两个不同操作中选择其一执行;switch 选择结构又称为多分支选择结构,在多个不同操作中作出选择。

C++ 提供了 while、do-while 和 for 3 种循环结构。

C++ 规定 if、else、switch、while、do 和 for 等都是关键字,这些关键字是该语言保留的,不能作为变量名等一般标识符。

4.2 表达式语句

表达式语句是 C++ 最基本的语句。在任何表达式之后加上分号,就是一个表达式语句。表达式语句的形式如下:

```
表达式;
```

表达式语句的功能是计算分号前表达式的结果,但该计算的结果并没有被再利用。需要特别说明的是,分号(;)是表达式语句的组成部分。

常用的表达式语句是赋值语句和具有返回值的函数调用语句,其中赋值语句就是赋值运算表达式构成的语句,而函数调用语句就是由函数调用构成的语句,这将在第 5 章中介绍。

注意:C++ 允许仅以分号“;”构成一个语句,这种语句称为空语句。空语句仅起标识作用,不做任何事情。

下面是一些表达式语句的示例:

```
a=a+3;                  //赋值语句
x=y=z=0;                //多重赋值语句
t=2,t+x+a;              //逗号表达式语句
z=i<j ?x : x+y;         //条件表达式语句
f1();                   //函数调用语句
x1=exp(x);              //函数表达式语句,计算 e^x
x2=pow(x, y);           //函数表达式语句,计算 x^y
```

其中,exp、pow 都是标准函数,其原型在 math. h 文件中。

4.3 顺序结构——复合语句

C++ 程序由若干语句构成。语句规定了对数据进行加工处理的顺序和方法,大部分语句按它在程序中的位置顺序执行,C++ 提供了复合语句来描述顺序结构。

复合语句也称为分程序或块,是包含在一对花括号内的任意的语句序列。有时也将用

花括号括起来的一段程序称为程序块。作为复合语句结束标志的右花括号后不需要分号。复合语句的语法如下：

```
<复合语句>→'{'<语句序列>'}'
<语句序列>→{<语句>}
```

在复合语句内可以有数据声明，被声明的数据仅在声明它的复合语句内起作用。

在复合语句中，还可以含有复合语句，这种情况称为复合语句的嵌套。

复合语句是典型的顺序结构，其中的语句按照书写的顺序依次执行，除非有其他控制语句改变了控制流程。执行流程如图 4-1 所示。

程序设计风格提示：*书写复合语句时，左右花括号要对齐，组成复合语句的各语句要相对花括号缩进一层并对齐。*

下面通过一个计算圆的周长和面积的程序来说明复合语句的使用。

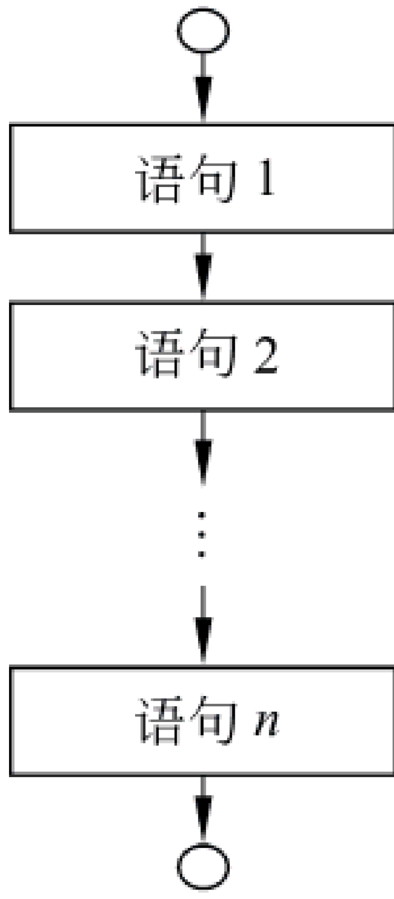


图 4-1　复合语句的执行流程

【例 4-1】 计算圆的周长和面积。

```
//ex4_1.cpp: 计算圆的周长和面积
#include<iostream.h>
#include<math.h>

#define PI   3.14
int main()
{
    double radius;
    double area;

    radius=3.0;
    {
        double perimeter;
        cout<<"The radius of the circle is : "<<radius<<"\n";
        perimeter=PI*2*radius;
        cout<<"The perimeter of the circle is : "<<perimeter<<"\n";
    }

    area=PI*radius*radius;
    cout<<"The area of the circle is : "<<area<<"\n";

    return 0;
}
```

程序输出为：

```
The radius of the circle is: 3
The perimeter of the circle is: 18.84
The area of the circle is : 28.26
```

在main函数中有一个复合语句,这个复合语句含有一个变量声明和一个赋值表达式语句,变量perimeter仅在这个复合语句中起作用。

4.4 选择结构

程序中除了顺序处理以外,有时还需要根据某些特定的条件决定对数据进行不同的操作,这就需要一种判断和选择的机制。为此,C++语言提供了3种类型的选择结构:if选择结构、if-else选择结构和switch选择结构。这3种结构可以实现单分支选择、双分支选择和多分支选择等功能。

4.4.1 if选择结构

像许多程序设计语言一样,C++也提供if选择语句。其一般形式为:

```
if  (<条件表达式>) <语句>
```

此结构表示:如果<条件表达式>的值为非0(true),即“真”,则执行指定的<语句>,然后按顺序执行if语句的后继语句;如果<条件表达式>的值为0(false),即“假”,则忽略<语句>,按顺序执行if语句的后继语句。执行流程如图4-2所示。

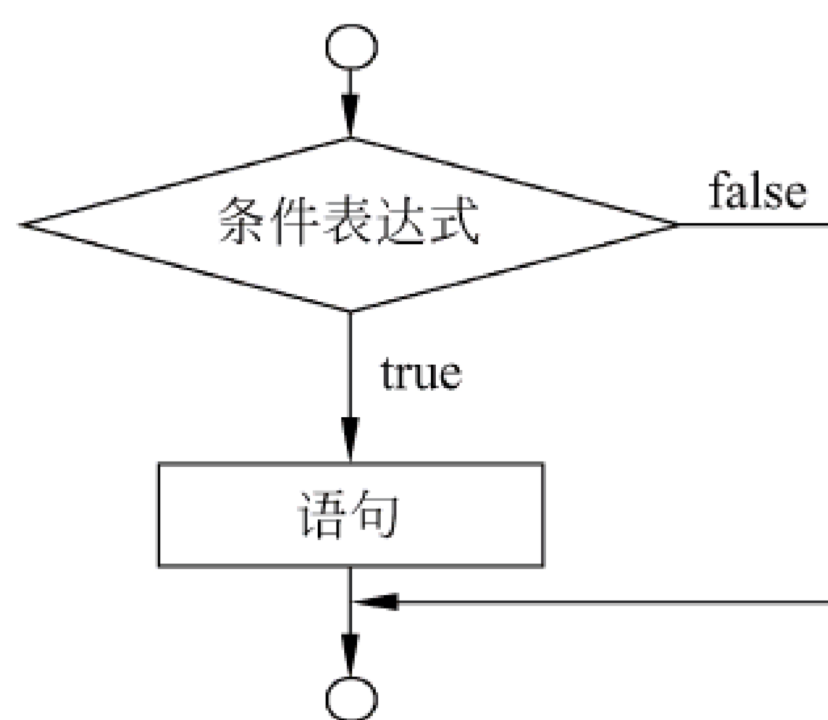


图4-2 if选择语句的执行流程

C++允许任意表达式充当条件表达式,并根据该表达式的值来确定条件的“真”或“假”。

程序设计风格提示:从语法上来说,整个if语句可以写成一行。如果测试表达式和语句都非常简单,那么整个语句可以写在一行内;否则,最好在测试表达式后换行,而且语句部分要相对if缩进一层,如果是复合语句更应如此。

从第2章的内容可知,任何合法的逻辑表达式都可以充当if语句的测试条件。事实上,也可以用任何表达式来作为测试条件:如果表达式的值为非0,就认为它为“真”;如果表达式的值为0,就认为它为“假”。

【例4-2】 if选择结构语句示例:根据输入的成绩输出不同的信息。

```
//ex4_2.cpp: if选择结构语句示例
#include<iostream.h>

#define PASS_GRADE          60
#define EXCELLENT_GRADE 90

int main()
{
    int score;

    //输入成绩
    cout<< "Please input your score:\n";
    cin>> score;
```

```
    //关系表达式作为测试条件
    if ( score< PASS_GRADE )
        cout<< "Sorry, you've failed! \n";
    if ( score> = PASS_GRADE )
        cout<< "Congratulation, you've passed! \n";
    //逻辑表达式作为测试条件
    if ( score> EXCELLENT_GRADE )
        cout<< "You've got an excellent score! \n";
    //复杂逻辑表达式作为测试条件
    if ( score> = PASS_GRADE && score< = EXCELLENT_GRADE )
        cout<< "Work harder! \n";
    return 0;
}
```

程序的运行情况如下：

```
Please input your score:
76
Congratulation, you've passed!
Work harder!
```

4.4.2 if-else 选择结构

if 选择结构只有在条件为真时，才执行指定的动作；否则，就跳过这个动作。实际应用中，经常需要根据测试条件的真假分别执行不同的处理，这种情况下 if 选择结构用起来就不自然了。为此，C++ 提供了一种双分支选择结构，即 if-else 选择结构，可以在条件为真或假时分别执行指定的不同动作。C++ 的 if-else 语句的一般形式为：

```
if  (<条件表达式>) <语句1>   else<语句2>
```

else 和<语句 2>称为 else 分支或 else 子句。上述结构表示：如果<条件表达式>的值为非 0，即“真”(true)，则执行<语句 1>，执行完<语句 1>后继续执行整个 if-else 语句的后继语句；如果<条件表达式>的值为 0，即“假”(false)，则跳过<语句 1>而执行<语句 2>，执行完<语句 2>后继续执行整个 if-else 语句的后继语句。也就是说，if-else 语句总是根据<条件表达式>的结果，选择<语句 1>和<语句 2>中的一个执行，执行完后，整个 if-else 语句就算执行完了。执行流程如图 4-3 所示。

程序设计风格提示：书写 if-else 语句时，if 和 else 要对齐，而分支的语句部分要缩进一层。

对于上节程序段，可以修改为：

```
if (grade> = 60)
    cout<< "Passed! \\n";
else
```

```
        cout<< "Failed! \\n";
```

这样实现的功能是：学生的成绩大于等于 60 时，输出"Passed!"；小于 60 时，输出"Failed!"。在这两种情况下，完成输出之后都会按顺序执行整个 if-else 语句的后继语句。下面的例子给出了用 if-else 选择结构实现根据成绩输出信息的功能。

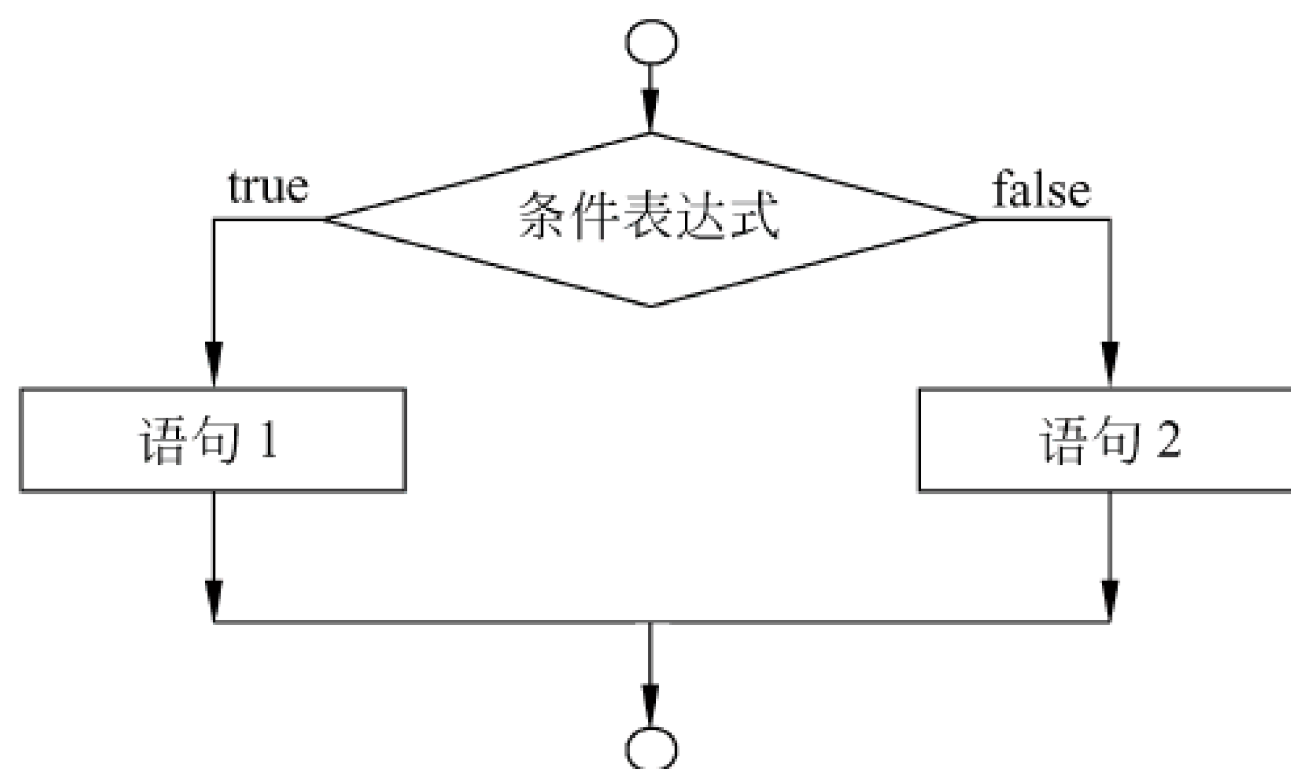


图 4-3　if-else 选择语句的执行流程

【例 4-3】　if-else 选择结构语句示例：根据用户输入的成绩，判断并输出是否通过了考试的信息。

```
//ex4_3.cpp: if-else 选择结构语句示例
#include<iostream.h>

int main()
{
    int score;
    //输入成绩
    cout<< "Please input your score:\n";
    cin>> score;
    //根据成绩输出考试是否通过的信息
    if(score>=60)
        cout<< "Congratulation, you've passed! \n";
    else
        cout<< "Sorry, you've failed! \n";
    return 0;
}
```

输入及格分数时程序的运行情况：

```
Please input your score:
85
Congratulation, you've passed!
```

输入不及格分数时程序的运行情况：

```
Please input your score:
55
Sorry, you've failed!
```

需要说明下面几点。

(1) <语句1>和<语句2>可以是复合语句,如果各分支有多于一条语句要执行时,必须使用花括号{和}把这些语句包括在其中,此时条件语句形式为:

```
if (表达式)
{
    语句体 1
}
else
{
    语句体 2
}
```

(2) if 选择结构与 if-else 选择结构中的分支可以是任何合法的语句,可以是简单的表达式语句如赋值语句,也可以是复合语句,甚至可以是 if 语句和 if-else 语句,这种情况称为 if 选择结构或 if-else 选择结构的嵌套。出现这类嵌套时,要特别注意 if 和 else 的匹配问题。例如,程序段:

```
if (x>20|| x<-10) if (y<=100 && y>x) printf("Good"); else printf("Bad");
```

有下面两种可能的解释。

```
if (x>20|| x<-10) {
    if (y<=100 && y>x)
      printf("Good");
}
else
    printf("Bad");
```

和

```
if (x>20|| x<-10) {
    if (y<=100 && y>x)
      printf("Good");
    else
       printf("Bad");
}
```

上边的解释是 else 与外层的 if 匹配,可视为在一个 if-else 语句中嵌套了一个 if 语句;下边的解释为 else 与内层的 if 匹配,可视为在一个 if 语句中嵌套了一个 if-else 语句。这种出现不同的解释的现象被称为二义性,不同的解释将导致不同的执行效果。然而程序只能采取一种执行方式。对于上述情况,C++ 规定:else 分支总是与最近的一个可以匹配 else 分支的 if 语句匹配,上例中的 else 与 if (y<=100 && y>x)相匹配,即 C++ 按照下边的解释执行。如果要让 else 与 if(x>20 || x<-10)相匹配,必须用复合语句,通过花括号来明确匹配关系。如下所示:

```
if (x>20|| x<-10)
```

```
{
    if (y<=100 && y>x)
    printf("Good");
}
else
    printf("Bad");
```

(3) 对于复杂连续的判断,可以采用阶梯式嵌套的 if-else-if 结构。其一般形式为:

```
if (<测试表达式 1>)
    <语句1>
else if (<测试表达式 2>)
    <语句2>
else if (<测试表达式 3>)
    <语句3>
       ⋮
else
    <语句n>
```

这种结构是从上到下逐个对条件进行判断,一旦发现条件满足就执行与它对应的语句,执行完后,整个 if-else-if 结构就执行完了,继续执行整个 if-else-if 结构的后继语句。从效果上看,相当于一旦满足某个条件就执行与它对应的语句,并跳过其他剩余分支语句。如果没有一个条件得到满足,则执行最后一个 else 后面的<语句 n>。最后这个 else 常起着默认处理的作用。类似地,如果某个条件的处理部分有多于一条语句要执行,必须使用{和}把这些语句包括在其中,构成一个复合语句。

【例 4-4】 输入 3 个数,找出其中最大的一个并显示出来。

设计思想:要选取 3 个数中的最大者,最简单的方法就是先拿第一个数与第二个数比较,用较大的数与第三个数作比较,较大的数就是三者中最大的数。

```
//ex4_4.cpp: 输入 3个数,找出其中最大的一个并显示出来
#include<iostream.h>

int main()
{
    int num1, num2, num3;
    int max;
    //输入数据
    cout<<"Please input three integers:\n";
    cin>>num1>>num2>>num3;
    //第一个数与第二个数作比较
    max=num1;
    if ( max<num2 )
        max=num2;
    //较大的数与第三个数作比较
    if ( max<num3 )
        max=num3;
```

```
    //输出最大数
    cout<<"The maximum integer is: "<<max<<endl;
    return 0;
}
```

程序的运行情况如下：

```
Please input three integers:
12 34 56
The maximum integer is: 56
```

【例 4-5】 输入 3 个数，按照由大到小的顺序将 3 个数显示出来。

设计思想：这是一个排序问题。排序的方法有很多，这里用最简单的方法将 3 个数排序：首先比较第一与第二个数的大小，较大的暂定为最大数，较小的暂定为最小数；然后将第三个数与暂定的最大者和最小者进行比较，最后确定大小顺序。

```
//ex4_5.cpp: 输入 3个数,按由大到小的顺序显示出来
#include<iostream.h>
int main()
{
    int num1, num2, num3;
    int max, min, mid;
    //输入数据
    cout<<"Please input three integers:\n";
    cin>>num1>>num2>>num3;
    //比较第一个数与第二个数的大小,较大的暂定为最大数,较小的暂定为最小数
    if ( num1<num2 )
    {
        min=num1;
        max=num2;
    }
    else
    {
        min=num2;
        max=num1;
    }
    //将第三个数与暂定的最大者和最小者进行比较
    if ( max<num3 )
    {
        mid=max;
        max=num3;
    }
    else if ( min>num3 )
    {
        mid=min;
```

```
            min=num3;
        }
        else
            mid=num3;
        //输出
        cout<<"The sorted integers are: ";
        cout<<max<<" "<<mid<<" "<<min<<endl;
        return 0;
}
```

程序的运行结果：

```
Please input three integers:
12 34 56
The sorted integers are: 56 34 12
```

4.4.3 switch选择结构

有时需要对多于两种可能性的情况进行分别处理，利用嵌套的if-else语句是一种实现方法。

【例4-6】 如果学生的考试成绩采取百分制(分数为0～100分的整数)评定，现要根据输入的学生成绩输出对应的等级评定(A、B、C、D和E)。

```
//ex4_6.cpp: 根据输入的学生成绩(百分制,分数为0~100分的整数)输出对应的等级评定(A、B、C、D、E)
#include<iostream.h>
int main()
{
        int score;
        //输入分数
        cout<<"Please input the score: \n";
        cin>>score;
        //判断并输出等级
        if ( score>=90 )
            cout<<"Grade is "<<'A'<<'.'<<endl;
        else if ( score>=80 )
            cout<<"Grade is "<<'B'<<'.' <<endl;
        else if ( score>=70 )
            cout<<"Grade is "<<'C'<<'.' <<endl;
        else if ( score>=60 )
            cout<<"Grade is "<<'D'<<'.' <<endl;
        else if ( score>=0 )
            cout<<"Grade is "<<'E'<<'.' <<endl;
        else
            cout<<"The score is illegal!"<<endl;
```

```
    return 0;
}
```

下面是两种可能的运行情况：

```
Please input the score:
90
Grade is A.
```

```
Please input the score:
77
Grade is C.
```

从这个程序可以看出，嵌套的if-else结构确实可以处理多分支的情况，但在表现各分支之间的关系上并不自然，而且容易出错。对于这类情况，C++提供了一种switch多分支选择结构，也称为开关语句。switch语句的格式为：

```
switch(<条件表达式>)
{
    case <常量表达式1>:
        <语句序列1>
    case <常量表达式2>:
        <语句序列2>
            ⋮
    case <常量表达式n>:
        <语句序列n>
    default:
        <语句序列n+1>
}
```

switch、case和default为C++的关键字；<条件表达式>是值为整型的表达式；每个case对应一个分支处理；default分支为默认处理分支；<常量表达式1>…<常量表达式n>都是值为整型常量的表达式；<语句序列1>…<语句序列$n+1$>都是一组语句，可以为空。

switch语句执行时，首先计算<条件表达式>得到一个整型的值，将该值与<常量表达式1>…<常量表达式n>的值逐个进行比较，如果与其中一个相等，则执行该常量表达式下的语句序列。注意，执行完该常量表达式对应的语句序列后，还将继续执行后续分支的处理语句序列，直到switch语句结束或者遇到转移指令；如果测试表达式的值不与任何一个常量表达式的值相等，则执行default分支后面的语句。switch语句的执行流程如图4-4所示。

需要注意以下几点。

（1）switch语句中测试表达式可以是整数值，也可以是字符。

（2）并不要求各分支处理要覆盖测试表达式的所有取值，可以省略一些case和default。

（3）每个case或default后的语句序列可以包含多条语句，不需要使用{和}括起来。

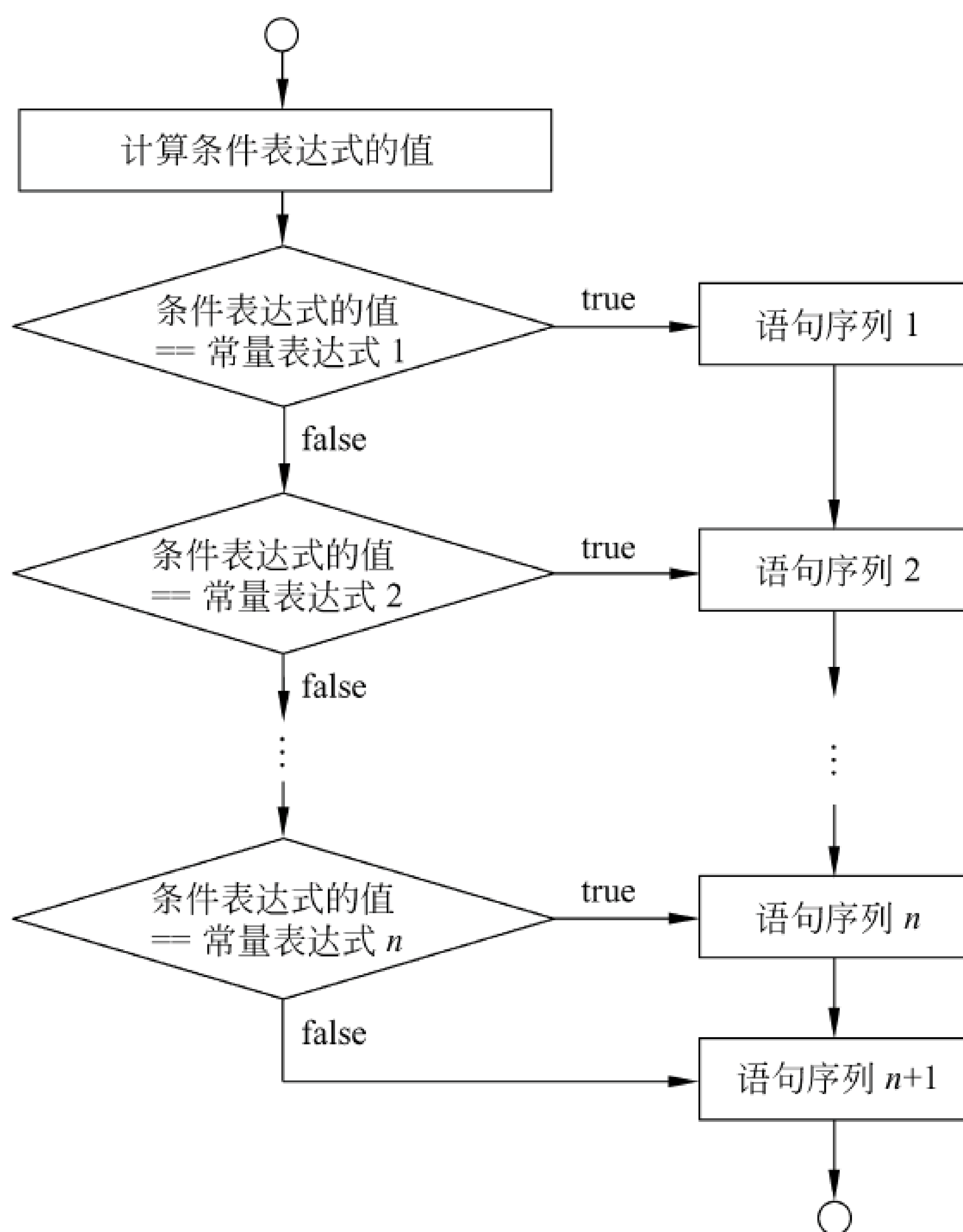


图 4-4　switch 语句的执行流程

程序设计风格提示：写 switch 语句时，switch 及测试表达式单独一行，各 case 分支和 default 分支要缩进一层并对齐，分支处理语句要相对再缩进一层，以体现不同层次的结构。

需要特别指出的是，当一个分支条件得到满足时，执行完该分支的语句序列后，还将继续执行后续分支的处理语句序列。

如果需要执行完一个分支后就结束整个 switch 语句，应该如何设计呢？C++ 提供了一种简便的方法——使用 break 语句。

break 语句是一种转移语句，该语句只能出现在 switch 结构和循环结构中。执行 break 语句，将导致终止包含该 break 语句的最内层的 switch 结构或循环结构，程序控制离开该 switch 结构或循环结构，直接执行其后继语句。程序模式是：

```
switch(<条件表达式>)
{
    case <常量表达式 1>:
        <语句序列 1>;
        break;
    case <常量表达式 2>:
        <语句序列 2>;
        break;
         ⋮
```

```
    case <常量表达式 n>;
        <语句序列 n>;
        break;
    default:
        <语句序列 n+1>;
}
```

其执行过程为：首先计算<条件表达式>，将得到的整型值与<常量表达式 1>…<常量表达式 n>的值逐个进行比较，一旦检测到与其中一个相等，则执行该常量表达式下的语句序列，执行完后，紧接着就执行 break 语句，离开该 switch 结构，不再执行后续分支的处理语句序列，直接转向整个 switch 语句的后继语句。

在程序设计过程中，可以灵活使用空的分支处理语句序列和 break 语句，以满足不同的处理需求。

【例 4-7】 用 switch 语句实现成绩转换：根据输入的学生成绩（百分制，分数为 0～100 分的整数）输出对应的等级评定（A、B、C、D 和 E）。

```
//ex4_7.cpp：根据输入的学生成绩（百分制，分数为 0~100 分的整数）输出对应的等级评定（A、B、C、D、E）
#include<iostream.h>

int main()
{
    int score;
    int scorePhrase;

    //输入分数
    cout<<"Please input the score: \n";
    cin>>score;
    if (score<0 || score>100)
    {
        cout<<"The score is illegal!"<<endl;
        return 0;
    }
    scorePhrase=score / 10;            //计算分数段

    //判断并输出等级
    switch ( scorePhrase )
    {
        case 10:
        case 9:
            cout<<"Grade is "<<'A'<<'.'<<endl;
            break;
        case 8:
            cout<<"Grade is "<<'B'<<'.' <<endl;
            break;
        case 7:
            cout<<"Grade is "<<'C'<<'.' <<endl;
```

```
                break;
            case 6:
                cout<<"Grade is "<<'D'<<'.'<<endl;
                break;
            case 0:
            case 1:
            case 2:
            case 3:
            case 4:
            case 5:
                cout<<"Grade is "<<'E'<<'.'<<endl;
                break;
            default:
                cout<<"The score is illegal!"<<endl;
        }
        return 0;
}
```

下面是几种可能的运行情况:

```
Please input the score:
85
Grade is B.

Please input the score:
55
Grade is E.

Please input the score:
120
The score is illegal!
```

4.5 循环结构

实际生活中,经常遇到需要对数据进行反复处理的情况。例如,为了统计某同学的考试总分,需要反复地将每门课程的成绩累加到总和中,直到所有的课程都处理完毕;全班所有同学的成绩可以通过反复的加法运算累加起来,然后求得平均成绩。为了支持对这类问题的处理,高级程序设计语言提供了多种循环控制结构,使得程序员能够方便地进行程序设计。C++有3种基本的循环控制结构语句:while语句、do-while语句和for语句。这3种循环结构具有不同的特点,分别适合不同的应用场合。

4.5.1 while循环语句

while循环语句的一般形式为:

```
while (<条件表达式>) <语句>
```

while 是 C++ 的关键字，是 while 语句开始的标记，＜条件表达式＞为一个合法的表达式，作为循环控制条件，＜条件表达式＞后面的语句是循环体。while 语句执行过程是：首先计算＜条件表达式＞的值，如果＜条件表达式＞的值为 0（即 false），则跳过指定的＜语句＞，执行整个 while 语句的后继语句；如果＜条件表达式＞的值为非 0（即 true），则执行指定的＜语句＞，执行完该语句后，再计算＜条件表达式＞的值，如果＜条件表达式＞的值仍然为非 0，则继续执行指定的＜语句＞，再进行测试……直到＜条件表达式＞的值为 0，再跳过指定的＜语句＞，结束整个 while 语句的执行，接着执行整个 while 语句的后继语句。执行流程如图 4-5 所示。

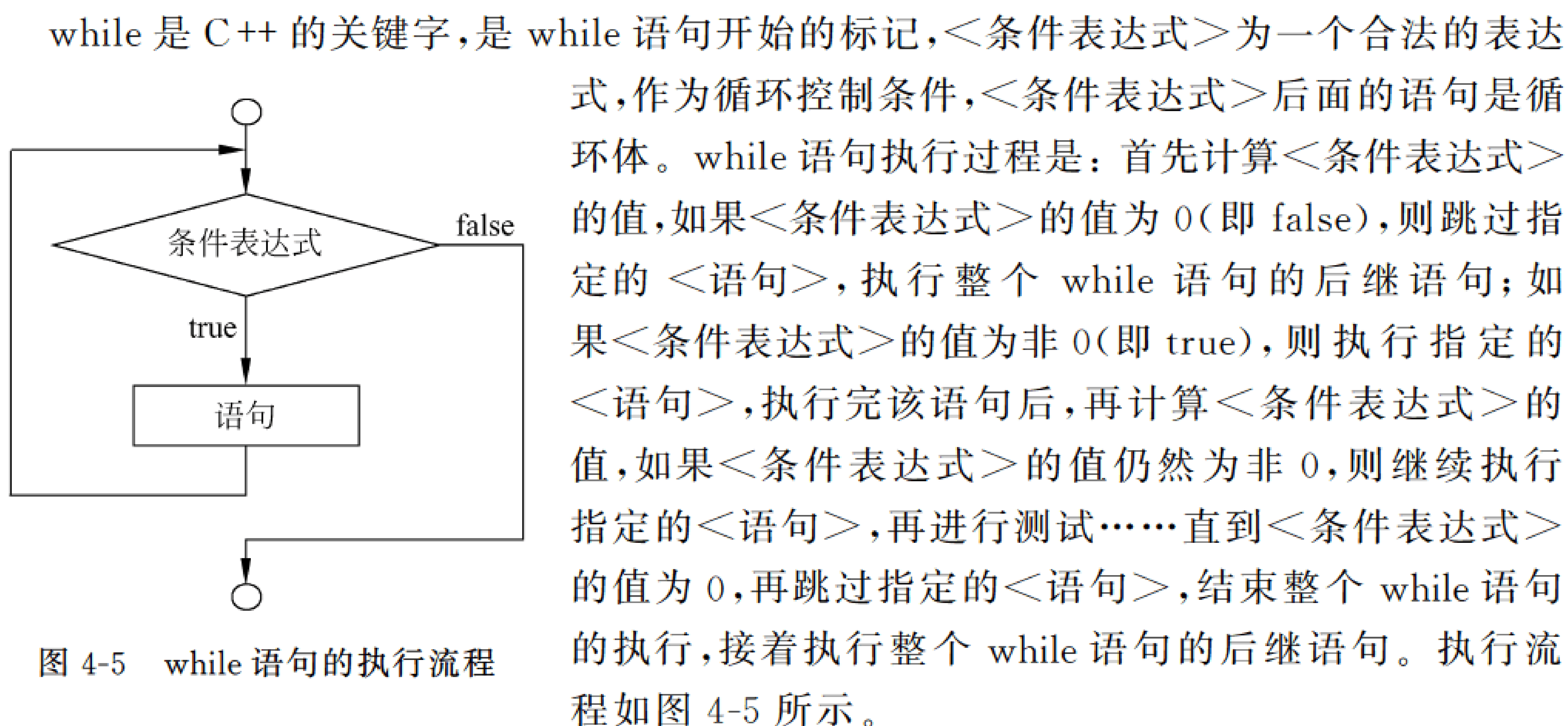


图 4-5　while 语句的执行流程

程序设计风格提示：书写 while 语句时，while 及测试表达式单独一行，循环体的语句要缩进一层。

如图 4-5 所示，while 语句的执行总是先测试＜条件表达式＞是否满足，然后才决定是否继续执行循环体。它特别适合非定数循环，即循环重复的次数在设计程序时不能确定，要依赖于运行时的具体情况。

【例 4-8】 利用 while 循环语句设计一个程序，该程序能够从键盘上接受任意个非负数（以负数作为数据输入的结束标记），并计算出这些数据的和。

设计思想：由于不知道待求和的数据的实际数目，所以无法预知循环执行的次数，只能根据输入的数据是否为负数来判断是否还有数据需要处理。流程图如图 4-6 所示。

```
//ex4_8.cpp: 计算任意个非负数的和
#include<iostream.h>
int main()
{
    //初始化
    int sum=0;
    int i;
    //输入数据到 i
    cout<<"Please input integers, using negative as the end
    mark: "<<endl;
    cin>>i;
    //求和
    while ( i>=0 )
    {
        sum=sum+i;
        //输入下一数据到 i
        cin>>i;
    }
    //输出
```

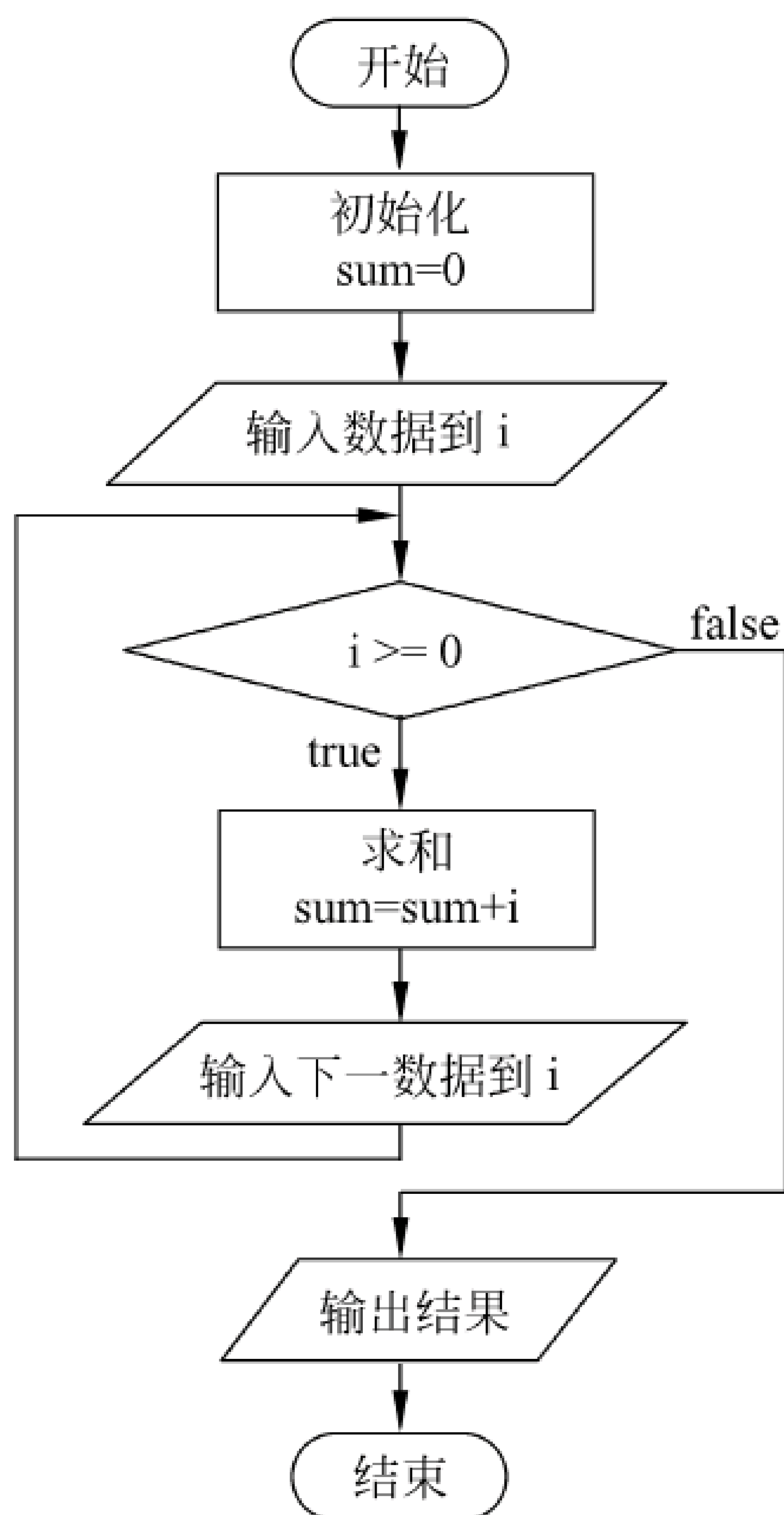


图 4-6　计算任意个非负数的和的流程图

```
    cout<<"The sum of the input integers is: "<<sum<<endl;
    return 0;
}
```

程序的运行结果:

```
Please input integers, using negative as the end mark:
12 34 56 78 -2
The sum of the input integers is: 180
```

注意事项如下。

(1) 在 while 循环体内也允许空语句。

(2) 可以有多层循环嵌套。

(3) 循环体的语句可以是复合语句体,此时必须用{和}括起来。

【例 4-9】 编写一个程序,输出 100 以内的斐波那契(Fibonacci) 数列,其定义如下:

$$a_1 = 1$$
$$a_2 = 1$$
$$a_n = a_{n-1} + a_{n-2} \quad n \geqslant 3$$

设计思想:由于不知道 100 以内的斐波那契数列元素的确切数目,因而只能先检查当前产生的元素的大小,如果小于等于 100,则继续产生后续的数列元素,直到元素的值大于 100,就终止循环。算法描述如下。

```
1. 初始化:a1 置为 1,a2 置为 1;
2. 计算并输出数列元素:
   如果 a1 和 a2 都小于等于 100,则反复执行
   2.1  输出当前的数列元素 a1 和 a2;
   2.2  计算下一组元素:
        2.2.1  a1=a1+a2;
        2.2.2  a2=a2+a1;
3. 判断并输出最后一组元素:
   3.1  如果 a1<=100, 输出 a1;
   3.2  如果 a2<=100, 输出 a2;
4. 结束。
```

```
//ex4_9.cpp: 输出的 100 以内的 Fibonacci 数列
#include<iostream.h>
int main()
{
    //初始化
    int a1=1, a2=1;
    //计算并输出数列元素
    while ((a1<=100) && (a2<=100))
    {
```

```
        cout<<a1<<"\t";
        cout<<a2<<endl;
        a1=a1+a2;
        a2=a2+a1;
    }
    //判断并输出最后一组元素
    if (a1<=100)
        cout<<a1<<"\t";
    if (a2<=100)
        cout<<a1<<endl;
    return 0;
}
```

程序执行结果：

```
1	1
2	3
5	8
13	21
34	55
89
```

另外，请读者注意理解程序的循环中计算数列元素的两个赋值语句的功能。

4.5.2 do-while 循环语句

C++ 还提供了另外一种循环结构——do-while 语句，其一般格式为：

```
do <语句> while (<条件表达式>);
```

do 和 while 都是 C++ 的关键字，do 和 while 之间的语句是循环体，＜条件表达式＞作为循环控制条件，整个 do-while 语句的最后是作为语句结束标志的分号。do-while 语句构成的循环与 while 语句构成的循环有所不同：它先执行循环中的＜语句＞，然后计算＜条件表达式＞的值，判断条件的真假，如果为 true，则继续循环；如果为 false，则终止循环，继续执行整个 do-while 语句的后继语句。因此，do-while 语句是一种出口控制的循环结构，其循环体至少要被执行一次，而 while 语句是入口控制的循环结构，其循环体未必会被执行。do-while 语句的执行流程如图 4-7 所示。

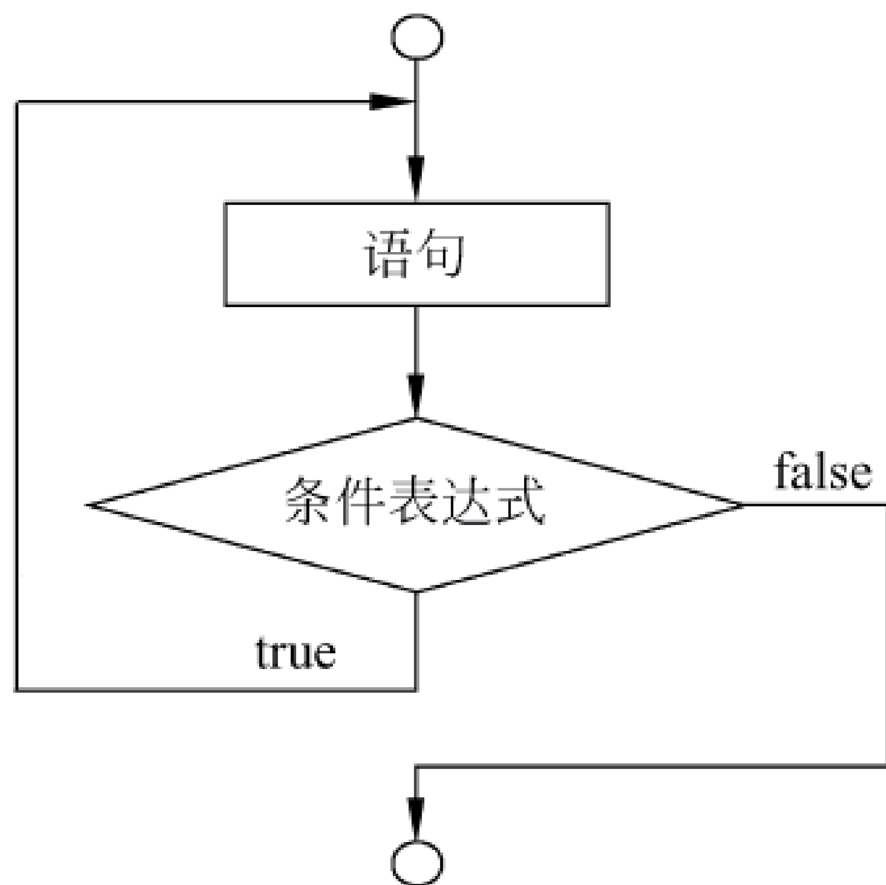


图 4-7 do-while 语句的执行流程

同样当循环体由多个语句组成时，要用{和}把它们括起来，组成一个复合语句。

程序设计风格提示：书写 do-while 语句时，do 单独占一行，循环体的语句要缩进一层，while 与 do 对齐。若循环体是复合语句，while 可接在复合语句右花括号后面。

【例 4-10】 编写一个程序,用下面的公式计算 π 的值,直到最后一项的绝对值小于 10^{-7}。

$$\frac{\pi}{4} \approx 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

程序如下:

```
//ex4_10.cpp: 计算 π 的值
#include<iostream.h>
#include<iomanip.h>
#include<math.h>

int main()
{
    //初始化
    double pi=1.0;
    double t=1.0;
    int n=1;

    //求数列元素
    do {
        t=1.0 / (2*n+1);
        if (n%2==1)
            t=-t;
        pi+=t;
        n++;
    } while ( fabs(t)>1e-7);

    pi=4*pi;
    cout<<"PI="<<setprecision(7)<<pi<<endl;

    return 0;
}
```

程序执行结果:

```
PI=3.141593
```

可以看出,do-while 语句与 while 语句功能相似,很多循环问题用两种方法都可以实现,但二者有明显的差异,实际编程时,应当根据具体情况选择最好的方案。

4.5.3 for 循环语句

在处理实际问题时,很多情况下可以预先知道循环应该重复的次数。例如,如果需要计算整数 1～100 的和,用 while 语句可以编写如下的代码进行实现:

```
int sum=0;
int i=1;
while ( i<=100 )
{
    sum=sum+i;
    i++;
}
```

而用 do-while 语句则实现如下：

```
int sum=0;
int i=1;
do
{
    sum=sum+i;
    i++;
} while ( i<=100 );
```

这两段程序有相似之处：变量 i 起到了循环计数的作用，该变量在循环体内的变化是有规律的，把这种变量称为循环控制变量。对于这类问题，C++ 专门提供一种语句——for 循环语句，又称计数循环。它的一般形式为：

```
for (<初始化语句>   [<条件表达式>]; [<增量表达式>] ) <语句>
```

for 是 C++ 的关键字，表示 for 循环语句的开始。＜初始化语句＞可以是任何合法的语句，＜条件表达式＞和＜增量表达式＞则可以由任何合法的表达式充当，其中＜初始化语句＞通常是一个赋值语句，用来给循环控制变量赋初值；＜条件表达式＞是一个能够转换成逻辑值的表达式，它决定什么时候退出循环，该表达式可以为空，这时逻辑值为 true；＜增量表达式＞定义了循环控制变量每循环一次后按什么方式变化，该表达式也可以为空，这时不产生任何计算效果。＜初始化语句＞可以是表达式语句或声明语句，应该以分号(;)结束。＜条件表达式＞和＜增量表达式＞之间用分号(;)分开。＜语句＞构成了循环体。

for 语句的功能是：首先计算＜初始化语句＞，然后计算＜条件表达式＞的值，如果该值为 false，则结束循环，跳过循环体的＜语句＞，转到整个 for 语句的后继语句继续执行；如果该值为 true，则执行循环体的＜语句＞，执行完循环体后，紧接着执行＜增量表达式＞，再计算＜条件表达式＞的值，如果该值为 true，则执行循环体的＜语句＞，执行＜增量表达式＞，再计算＜条件表达式＞进行测试……，直到＜条件表达式＞的值为 false，则结束循环，跳过循环体的＜语句＞，继续执行整个 for 语句的后继语句。for 语句流程如图 4-8 所示。

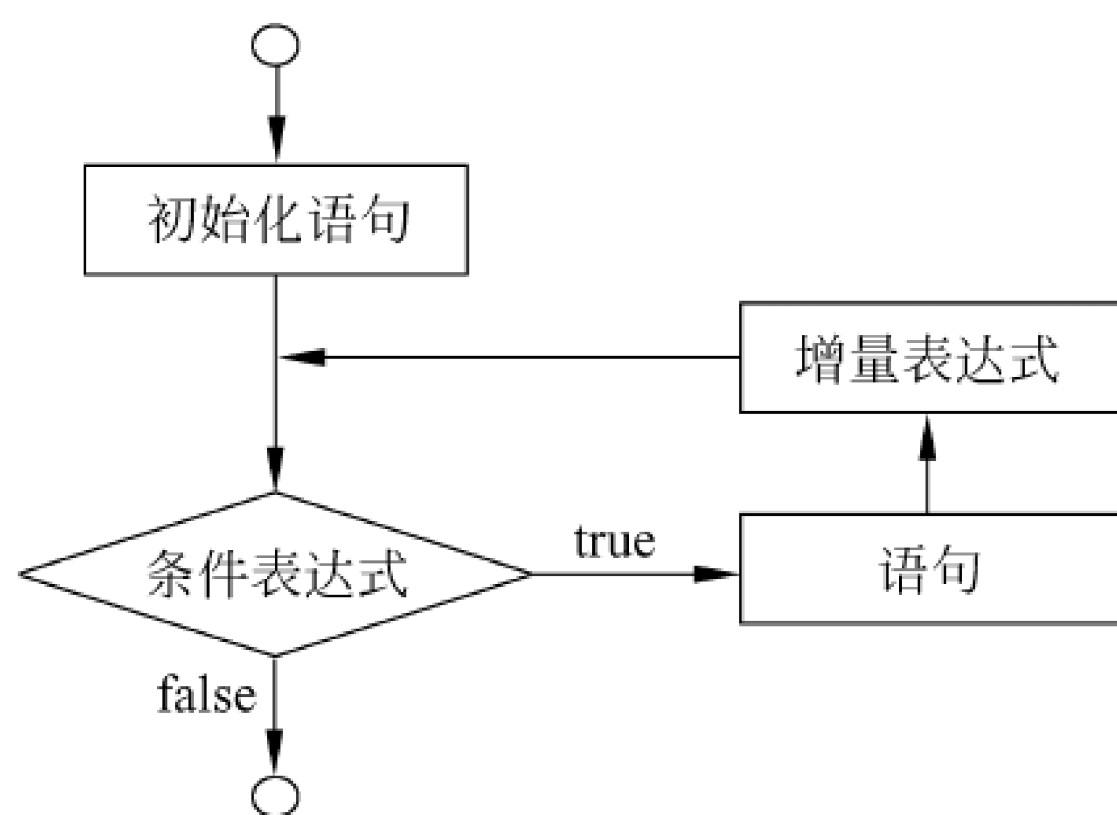


图 4-8　for 语句的执行流程

程序设计风格提示：书写 for 循环语句时，循环体的＜语句＞相对于 for 缩进一层。

下面的例子显示 for 循环结构中改变控制变量的方法。在每个例子中，给出了相应的 for 循环结构的首部。注意循环中控制变量的变化与循环终止条件的关系。

(1) 将控制变量从 1 变到 100，增量为 1。

```
for (i=1; i <=100; i++)
```

(2) 将控制变量从 100 变到 1，增量为－1。

```
for (i=100; i>=1; i--)
```

(3) 控制变量的变化范围为 7～77，增量为 7。

```
for (i=7; i<=77; i+=7)
```

(4) 控制变量的变化范围为20～2，增量为－2。

```
for (int i=20; i>=2; i-=2)
```

(5) 按所示数列改变控制变量值：99、88、77、66、55、44、33、22、11、0，增量为－11。

```
for (int j=99; j>=0; j-=11)
```

(6) 控制变量i和j共同进行循环控制，i从1变到99，j从2变到100，增量均为2。

```
for (int  i=1, j=2; i<=99 && j<=100; i+=2, j+=2)
```

需要说明以下几点。

(1) for循环中的＜语句＞可以为复合语句。

(2) for循环中的＜初始化语句＞可以是空语句(即仅以分号构成的语句)，表示不对循环控制变量赋初值。＜条件表达式＞和＜增量表达式＞都是可选项，可以缺省，但分号(;)不能省略。若省略＜条件表达式＞，如果没有其他处理，将导致死循环；若省略＜增量表达式＞，则每次循环体执行完后，不会对循环控制变量进行操作，这时语句体中应有修改循环控制变量的语句。

(3) 可以在for循环的＜初始化语句＞中声明变量(如上面的最后3个例子)，这些变量只在该for循环结构中有效，离开了该for结构，变量就无效了。

(4) for循环可以有多层嵌套，也就是说，for语句的循环体中可以包含其他语句，包括for语句。

【例4-11】 利用for循环语句计算并输出1～100的和。

```
//ex4_11.cpp: 利用 for语句计算 1至 100的和
#include<iostream.h>

int main()
{
    //初始化
    int sum=0;

    //求和
    for (int i=1; i<=100; i++)
        sum+=i;

    //输出
    cout<<"The sum of 1 to 100 is: "<<sum<<endl;

    return 0;
}
```

程序执行结果：

```
The sum of 1 to 100 is: 5050
```

【例4-12】 利用for循环语句计算并输出1～100中奇数和偶数的和。

```
//ex4_12.cpp：利用 for语句计算 1至 100中奇数和偶数的和
#include<iostream.h>
int main()
{
    //初始化
    int sumOfOddNum=0;
    int sumOfEvenNum=0;
    //求和
    for (int odd=1 , even=2; odd<=99 && even<=100; odd+=2, even+=2)
    {
        sumOfOddNum+=odd;
        sumOfEvenNum+=even;
    }
    //输出
    cout<<"The sum of odd numbers between 1 to 100 is: "<<sumOfOddNum<<endl;
    cout<<"The sum of even numbers between 1 to 100 is: "<<sumOfEvenNum<<endl;
    return 0;
}
```

程序执行结果：

```
The sum of odd numbers between 1 to 100 is: 2500
The sum of even numbers between 1 to 100 is: 2550
```

灵活利用 for 循环语句还可以实现许多有趣的功能，例如，可以设计一个程序画出如图 4-9 所示的等腰三角形。

【例 4-13】 利用 for 循环语句输出图 4-9 的等腰三角形。

设计思想：图 4-9 中的等腰三角形是有规律的，该三角形总共有 4 行，第一行有 1 个 *，第二行有 3 个 *，第三行有 5 个 *，第四行有 7 个 *，而且每行第一个 * 的起始位置分别是 4、3、2、1，即前导空格数目分别为 3、2、1、0。总结上述规律如下：第 i 行有 2*i−1 个 *，前导空格数目是 4−i。按照这一规律，利用循环嵌套输出每一行，就可得到整个三角形。

```
   *
  ***
 *****
*******
```

图 4-9 等腰三角形

```
//ex4_13.cpp：利用 for循环语句输出输出等腰三角形
#include<iostream.h>
#define LINE_NUM   4
int main()
{
    int line;                          //行控制变量
    int blankNum;                      //前导空格控制变量
    int starNum;                       //'*'控制变量
    //输出等腰三角形
    for ( line=1; line<=LINE_NUM; line++)
    {
```

```
            //输出前导空格
            for (blankNum=1; blankNum<=LINE_NUM-line; blankNum++)
                cout<<' ';
            //输出'*'
            for (starNum=1; starNum<=2*line-1; starNum++)
                cout<<'*';
            //换行
            cout<<endl;
        }
        return 0;
}
```

程序中将行数定义为符号常量,使得程序更容易维护。例如,如果需要输出行数为10的三角形,只要将LINE_NUM定义为10,而执行代码无须变动。

while结构、do-while结构和for结构具有不同的特点,可以证明,任何循环结构都可以用while结构来实现,do-while结构和for结构的功能用while结构也能完成。实际使用时,应当根据具体问题的特点选择最合适、最自然的结构来实现,以保证良好的程序设计风格。

程序设计技巧提示: *如果不能预先确定循环的次数,则应当选择合适的特征数据作为循环控制的条件,这时使用while结构和do-while结构较合适;如果事先可以确定循环的次数,或者存在某数据随循环有规律的变化,这时宜采用for结构。*

4.6 控制转移语句

控制转移语句的功能是无条件改变程序的执行顺序。早期的程序设计主要通过无条件goto语句实现控制的跳转。使用goto语句确实可以使程序员随心所欲地控制程序的流程。但是,如果滥用goto会导致程序的正确性难以控制,为程序的调试和维护带来很大的困难。因此,提倡尽可能少用goto语句。计算机科学家甚至证明了任何程序可以只用3种控制结构即顺序结构、选择结构和循环结构来实现,无需goto语句。但该原则也并不是绝对的,不可否认,适当地使用控制转移语句确实能为程序设计带来好处。C++中可以使用4种控制转移语句:break、continue、goto和return语句。其中,break和continue语句不同于goto语句,它们与控制结构相关,带有结构化的色彩,对程序结构的破坏性比goto要小,所以这两种控制语句使用较多。return语句实现从函数调用中返回。此外,C++改变程序执行顺序还可用exit和abort等标准函数,它们的原型在stdlib.h文件中说明。

4.6.1 break语句

break语句的语法非常简单:

```
break;
```

break语句通常用在循环语句和switch语句中。当break用于switch语句的分支处理

代码中时，可使程序控制跳出 switch 结构而执行该 switch 语句的后继语句。break 在 switch 中的用法已在介绍 switch 结构语句时出现，这里不再举例。

当 break 语句用于 while、do-while、for 循环结构语句中时，可使程序终止包含该 break 语句的最内层的循环，而执行该循环语句的后继语句。通常 break 语句总是与 if 语句联在一起，即满足条件时便跳出循环。

【例 4-14】 编写一个程序，判断输入的整数是否是素数，并输出判断结果(输入的整数小于或等于 0 表示结束)。

设计思想：素数只有 1 和自身两个因子，对于任何自然数，只要依次判断 2～n－1 之间是否存在 n 的因子就可以了(实际上还有更高效的方法，请思考)。

```
//ex4_14.cpp: 判断输入的整数是否是素数
#include<iostream.h>

int main()
{
    int n;
    bool isPrime;

    cout<<"Please enter an integer: ";
    cin>>n;
    while (n>0)
    {
        isPrime=true;
        for (int i=2; i<=n-1; i++)
            if (n%i==0)
            {    //i是 n的因子
                isPrime=false;
                break;
            }

        if (isPrime)
            cout<<n<<" is a prime number."<<endl;
        else
            cout<<n<<" is not a prime number."<<endl;

        cout<<"Please enter an integer: ";
        cin>>n;
    }

    return 0;
}
```

程序执行结果：

```
Please enter an integer: 123
123 is not a prime number.
Please enter an integer: 71
71 is a prime number.
```

```
Please enter an integer: 0
```

应该注意下面两点。

(1) break 语句对 if-else 条件语句不起作用。

(2) 在多层循环中,break 语句终止的是包含该 break 语句的最内层的循环,即只向外跳一层。

4.6.2 continue 语句

continue 语句的语法如下:

```
continue;
```

continue 语句只用在 while、do-while 和 for 3 种循环结构中,其功能是跳过该循环体中剩余的语句而强行开始下一次循环。continue 语句常与 if 条件语句一起使用,用来跳过某些特殊的情况或加速循环处理。在 while 或 do-while 结构中,循环条件测试在执行 continue 语句之后立即求值并作判断,而在 for 结构中,则先执行递增表达式,然后进行循环条件测试。

下面例子给出了不同循环结构中 continue 语句的使用示例,注意思考为什么 while、do-while语句内执行 continue 语句之前要对控制变量进行自增。

【例 4-15】 循环中 continue 语句的使用示例。

```
//ex4_15.cpp: 循环中 continue 语句的使用
#include< iostream.h>
int main()
{
    int x;
    //while 循环中 continue 语句的使用
    x=1;
    while ( x<=10 )
    {
        if ( x==5 )
        {
            x++;
            continue;                    //x 为 5 时,跳过,开始下一次循环
        }
        cout<< x<< " ";
        x++;
    }
    cout<< endl;
    //do-while 循环中 continue 语句的使用
    x=1;
    do
    {
```

```
        if ( x==5 )
        {
            x++;
            continue;                    //x为5时,跳过,开始下一次循环
        }
        cout<<x<<" ";
        x++;
    } while ( x<=10 );
    cout<<endl;
    //for循环中 continue语句的使用
    for ( x=1; x<=10; x++)
    {
        if ( x==5 )
            continue;                    //x为5时,跳过,开始下一次循环
        cout<<x<<" ";
    }
    cout<<endl;
    return 0;
}
```

程序执行结果:

```
1  2  3  4  6  7  8  9  10
1  2  3  4  6  7  8  9  10
1  2  3  4  6  7  8  9  10
```

4.6.3 goto 语句

goto 语句是一种无条件转移语句,使用格式为:

```
goto <标号>;
```

<标号>是 C++ 中一个有效的标识符,这个标识符加上一个冒号(:)一起出现在某个语句的前面,以标识该语句。执行 goto 语句时,程序将跳转到该标号处,从其后的语句开始继续执行。标号必须与 goto 语句同处于一个函数中,但可以不在一个循环层中。通常 goto 语句与 if 条件语句联用,当满足某一条件时,程序跳到标号处运行。

在结构化编程中,提倡尽可能不用 goto 语句,主要因为它将破坏程序的结构,使程序层次不清晰、不易读,但在某些情况下,适当使用 goto 语句也有利于程序的可读性。如例 4-16 中的多层嵌套需要退出时,用 goto 语句则比较合理。请思考,使用 break 语句如何实现该功能。

【例 4-16】 设计一个程序,按照 5 个整数一组输入数据,每组数据分别求和并输出结果,遇到负数结束。

```
//ex4_16.cpp: 循环中 goto语句的使用
```

```
#include<iostream.h>
int main()
{
    int i, j;
    int x;
    int sum;
    //for循环中 continue语句的使用
    for ( i=1; i<=10; i++)
    {
        sum=0;
        cout<<"Please enter 5 integers: "<<endl;
        for ( j=1; j<=5; j++)
        {
            cin>>x;
            if ( x<0 )
                goto end;                    //x为负数时,跳出
            sum+=x;
        }
        cout<<"Sum is: "<<sum<<endl;
    }
end:
    cout<<"Jumped out of loop."<<endl;
    cout<<"Current sum is: "<<sum<<endl;
    return 0;
}
```

程序执行结果:

```
Please enter 5 integers:
1 2 3 4 5
Sum is: 15
Please enter 5 integers:
6 7 8 9 10
Sum is: 40
Please enter 5 integers:
11 12 -3
Jumped out of loop.
Current sum is: 23
```

4.6.4 return 语句

return 语句在函数中使用。程序执行到 return 语句时,立刻从被调用函数中无条件返回到调用函数。在主函数中遇到 return 语句,将导致整个程序结束。

4.7 结构化程序设计方法

第1章已提及在计算机上求解问题的基本过程和算法的概念，本节将进一步介绍结构化程序设计方法。

4.7.1 结构化程序设计思想

1. 结构化程序的构成

前面学习了顺序结构、选择结构和循环结构3种基本控制结构。从结构化的观点来看，所有程序都是根据程序所需的算法组合这3种控制结构而成。用C++编写的结构化程序是由许多函数组成的，每个函数只有一个入口和一个出口，并且仅由上面3种结构构成，这种程序称为结构化程序。

本章前面几节中给出的C++ 3种基本控制结构流程图中，每个流程图都使用了两个小圆圈，一个是控制结构入口点，另一个是控制结构出口点。从这些图中可以看出，这些控制结构都是单入口/单出口的控制结构，这种结构使程序更容易建立，只要将一个控制结构的出口与另一个控制结构的入口连接，即可组成结构化程序。这种控制结构的并列连接方法称为控制结构堆叠(control structure stacking)，还有另一种控制结构连接方法，称为控制结构嵌套(control structure nesting)。

2. C++的结构化程序形成的规则

假设在流程图中用矩形框表示任何操作，包括输入输出，那么在C++中形成结构化程序的规则如下所述。

(1) 从"最简单的流程图"开始，如图4-10所示。

(2) 将其中某个矩形框(操作)换成两个顺序矩形框(操作)，即控制结构堆叠。

(3) 将其中某个矩形框(操作)换成某个控制结构(如if、if-else、switch、while、do-while或for结构)，即控制结构嵌套。

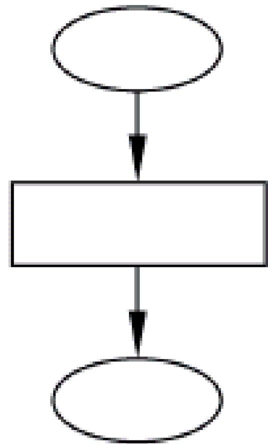


图4-10 最简单的流程图

(4) 可按任何顺序多次重复规则(2)和规则(3)。

利用上述规则总是可以得到整洁的结构化流程图。例如，对最简单的流程图重复采用规则(2)即可得到包含许多顺序放置矩形框的流程图(如图4-11所示)。这是一种堆栈控制结构，因此规则(2)被称为堆栈规则。

规则(3)被称为嵌套规则。对最简单的流程图重复采用规则(3)即可得到包含嵌套控制结构的流程图。例如，图4-12中，首先将最简单的流程图中的矩形框换成双分支选择结构(if-else)。然后再对双分支选择结构中的两个矩形框采用规则(3)，将两个矩形框中的一个变成一个双分支选择结构，另一个变成while循环结构，每个结构周围的虚线框表示最初的简单流程图中被替换的矩形框。

应用规则(4)可以产生更大、更复杂且层次更多的嵌套结构，构成各种可能的结构化流程图，从而实现各种可能的结构化程序。从上面的例子可以看出，应用上述规则的过程就是自顶向下、逐步求精的设计过程。

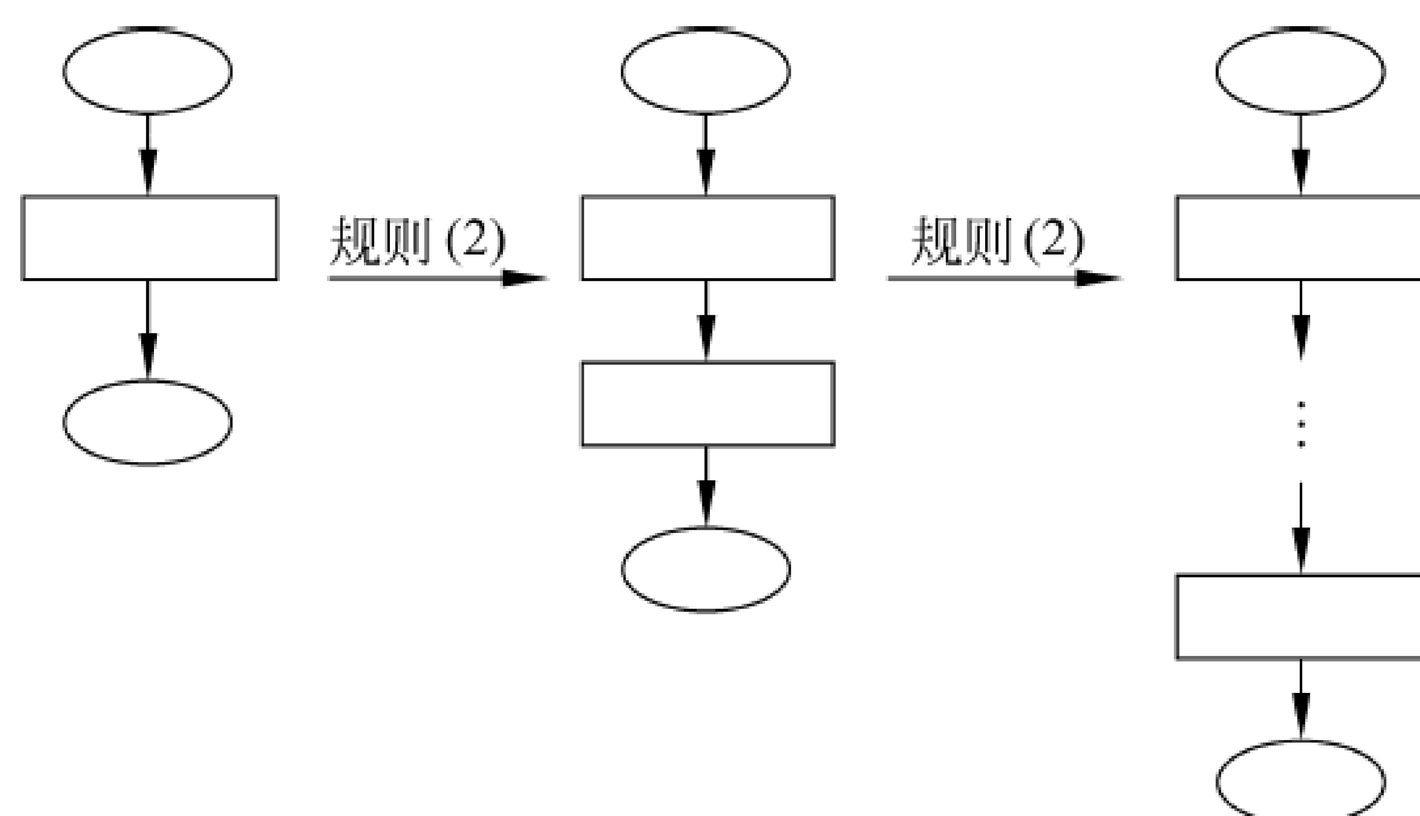


图 4-11　最简单的流程图重复采用规则(2)

结构化方法的精髓在于只需使用简单的单入/单出块、通过两种简单组合的方法，就可以实现程序的设计。图 4-13 显示出采用规则(2)的堆栈构件块和采用规则(3)的嵌套构件块。

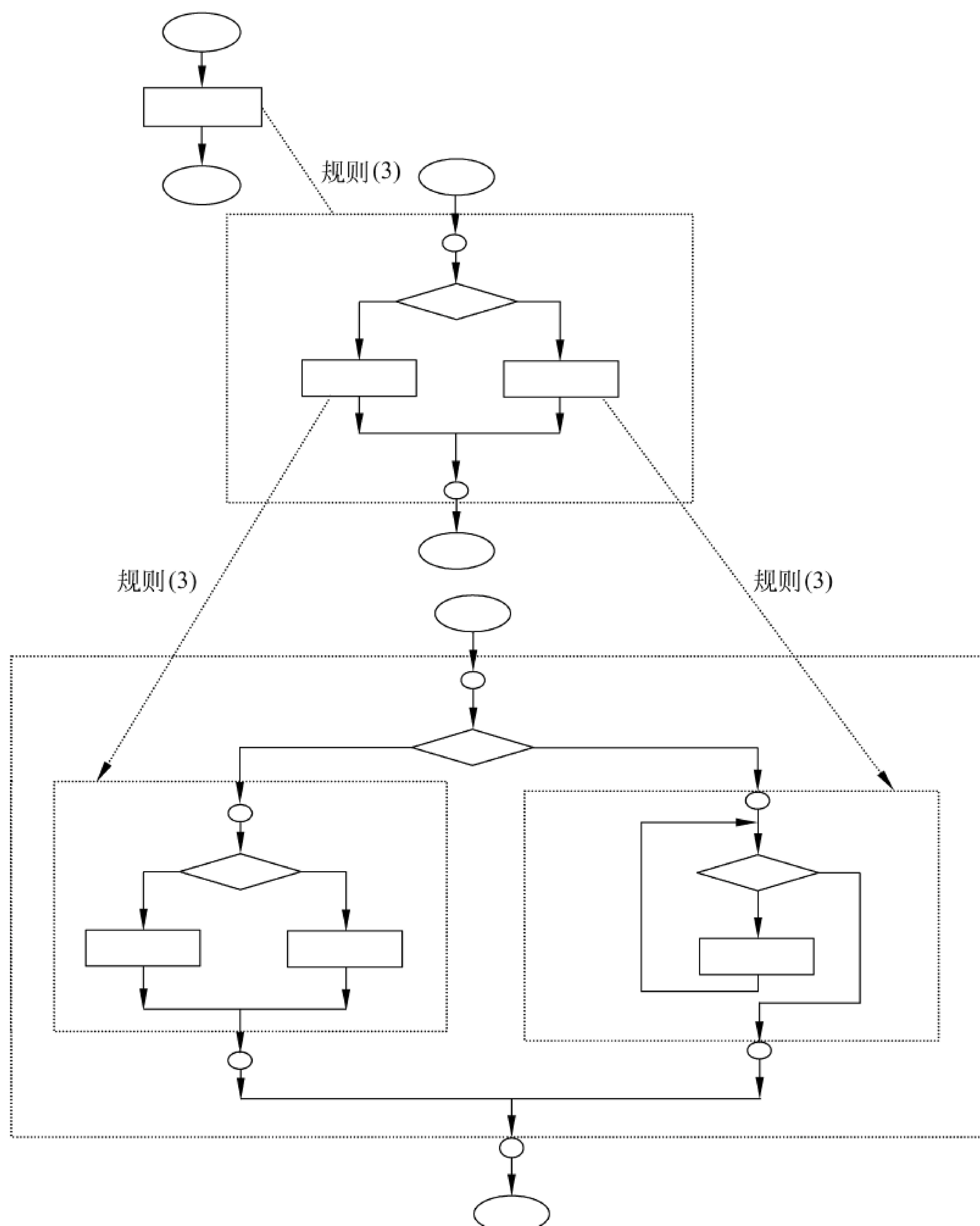
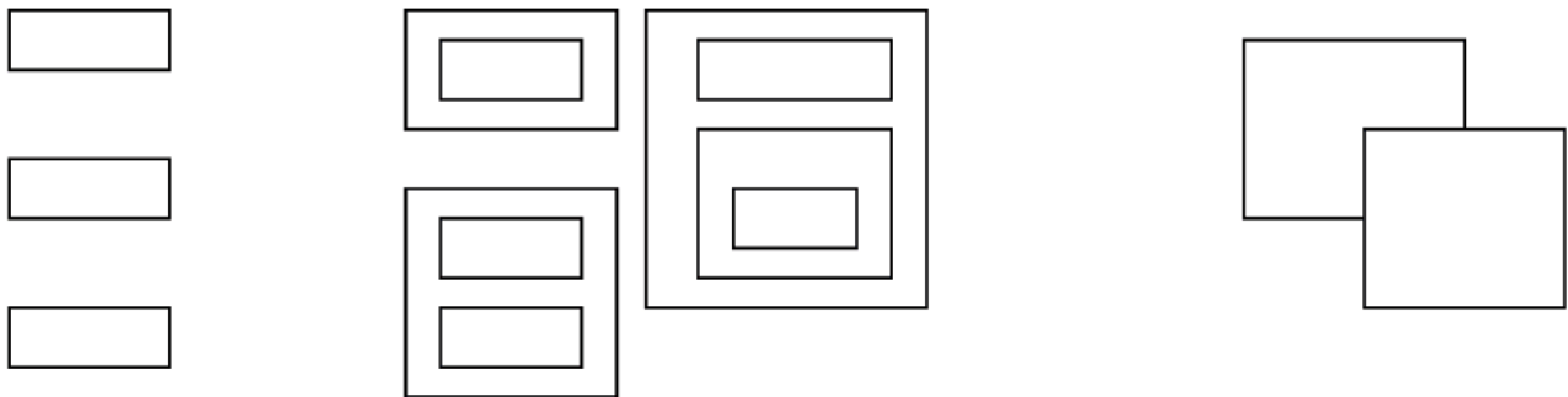


图 4-12　最简单的流程图重复采用规则(3)

图中还显示了结构化流程图中不能出现重叠构件块(因为结构化程序设计要消除 goto 语句)。



(a) 堆栈构件块　　(b) 嵌套构件块　　(c) 结构化流程图中不能出现重叠构件块

图 4-13　堆栈、嵌套和重叠构件块

综上所述,结构化程序设计的思想包括以下几方面的内容。

(1) 程序由一些基本结构组成。任何一个大型的程序都可以由 3 种基本结构所组成,由这些基本结构顺序地构成了一个结构化的程序。这 3 种基本结构为顺序结构、选择结构和循环结构。

(2) 一个大型程序应该按照功能分割成一些功能模块,并把这些模块按层次关系进行组织。

(3) 在程序设计时应采用自顶向下、逐步求精的实施方法。

按结构化程序设计方法设计出的程序具有如下优点:结构良好,各模块间的关系清晰简单,每一模块内都由基本单元组成。这样设计出的程序清晰易读,可理解性好,容易设计,容易验证其正确性,也容易维护。同时,由于采用自顶向下、逐步细化的实施方法,能有效地指导和组织程序设计活动,有利于软件的工程化开发。

4.7.2　结构化程序设计原则

结构化程序设计就是把一个应用程序划分成若干个基本结构,在编写程序代码时,各结构独立编写,最后统一成为一个整体。结构化程序设计要遵循的原则是:自顶向下、逐步求精、模块化和限制使用 goto 语句。

1. 自顶向下、逐步求精

所谓"自顶向下、逐步求精",是指程序设计时,应先考虑总体,后考虑细节;先考虑全局目标,后考虑局部目标。也就是说,先设计第一层(即顶层)问题的求解方法,然后步步深入,设计一些比较粗略的子目标作为过渡,再逐层细分,直到整个问题可用程序设计语言明确地描述出来为止。

采用自顶向下、逐步求精的原则进行程序设计时,为了设计一个复杂程序,首先必须作出对问题本身的确切描述,并对问题解法作出全局性决策,把问题分解成相对独立的子问题,再以同样的方式对每个子问题进一步精确化,直到获得计算机能理解的程序为止。

2. 模块化

任何一个大系统都可以按子结构之间的疏密程度分解成较小的部分,每部分称为模块,每个模块完成一定问题的求解。整个程序是由层次的逐级抽象的诸模块组成。所谓"模块化",是指对于一个复杂问题来说,它肯定是由若干稍简单的问题构成;为解决这个复杂问

题，要把它分解成若干稍小的、简单的部分。这一过程称为“模块划分”。模块化与自顶向下、逐步求精紧密联系。模块划分的基本要求如下。

（1）模块的功能在逻辑上尽可能单一、明确化、一一对应。

（2）模块之间的联系及影响尽可能少，必要的联系须加以明确说明，尽量避免传递控制信号，仅限于传递处理对象。

（3）每个模块的规模不能够过大，以使其本身易于实现。

3. 限制使用 goto 语句

结构化程序设计中，要尽可能限制使用无条件转移语句(goto 语句)。因为它将破坏程序的结构化逻辑，使程序模块间的界面模糊，降低了程序的可读性，直接影响程序的质量。除非使用 goto 语句能明显降低程序设计的难度，并且对程序结构的破坏性较小，一般情况下建议不要使用 goto 语句。

4.7.3 结构化程序设计示例

下面用具体例子来说明如何采用自顶向下、逐步求精的结构程序设计方法进行程序设计。

【例 4-17】 编写一个程序，能够从键盘读入一个正整数，输出从 2 到该正整数之间的所有素数。

首先分析这个问题，明确需求：

- 读入一个正整数。
- 输出从 2 到该正整数之间的所有素数。

下面采用自顶向下、逐步求精的方法来设计本题的算法，算法采用文字描述和框图两种方式描述。

问题的顶层描述是：

输入一个正整数，输出从 1 到该正整数之间的所有素数。

顶层描述是对整个程序功能的完整、精简的描述，要在顶层描述中明确程序设计的需求，作为逐步求精的基础。

接着，对顶层描述做第一步求精，得到一个顺序结构(见图 4-14)。

文字描述：
1. 输入一个正整数 limit；
2. 输出从 2 到 limit 之间的所有素数；
3. 结束。

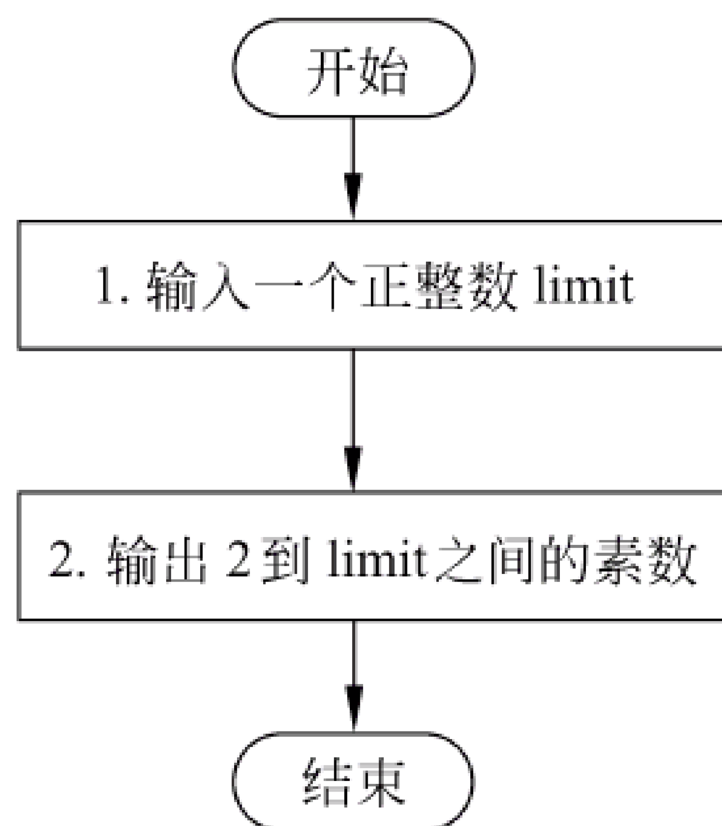


图 4-14 第 1 步求精

但是细化成这几个步骤，还不足以和程序结构对应起来，因此还需要进一步求精。

首先考虑第 1 步“输入一个正整数”。这一步看上去很简单，似乎用程序的输入语句就可实现，但是考虑到用户可能输入一个小于等于 0 的数据，为了增加程序的健壮性，还需要

增加一些处理功能：判断用户输入的数是否大于0，是则继续；否则，报错并结束程序。因而将算法细化如下(见图4-15)。

文字描述：
1. 输入一个正整数；
   1.1 读入一个整数limit；
   1.2 判断limit是否小于等于0，若是则输出报错信息，结束程序；
2. 输出从2到limit之间的所有素数；
3. 结束。

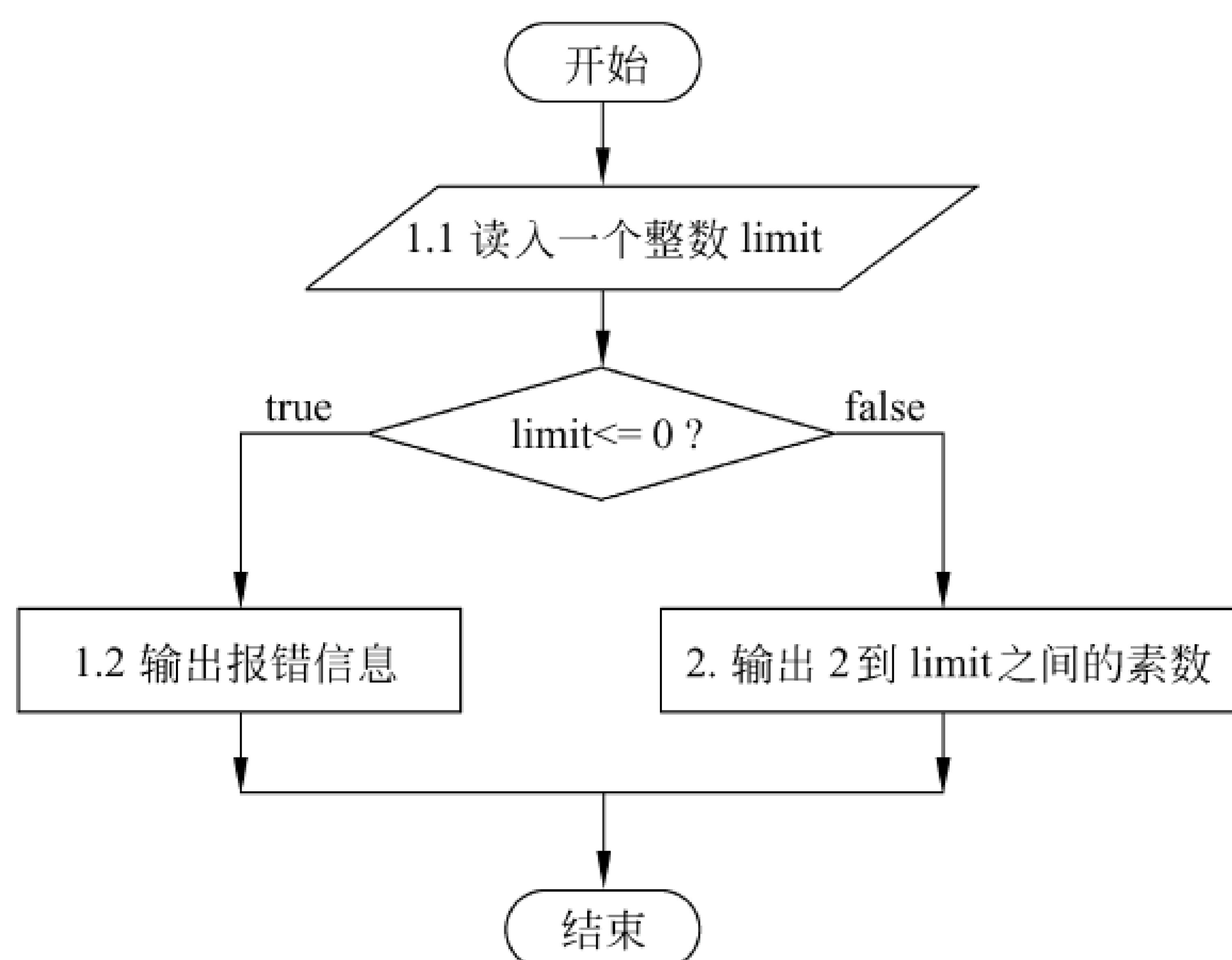


图4-15　对第1步“输入一个正整数”求精后

下面考虑程序的核心部分——第2步“输出从2到该正整数之间的所有素数”。本部分需要对2～limit的所有整数逐个进行判断，若是则输出该数；否则不输出。于是，对算法的第2步使用循环结构进行求精，如图4-16所示。

文字描述：
1. 输入一个正整数；
   1.1 读入一个整数limit；
   1.2 判断limit是否小于等于0，若是则输出报错信息，结束程序；
2. 输出从2到该正整数之间的所有素数；
   2.1 循环控制变量i置2；
   2.2 判断i <= limit，
      2.2.1 若是，则判断i是否为素数，若是则输出该数；
      2.2.2 否则，转3；
   2.3 i自增，转2.2；
3. 结束。

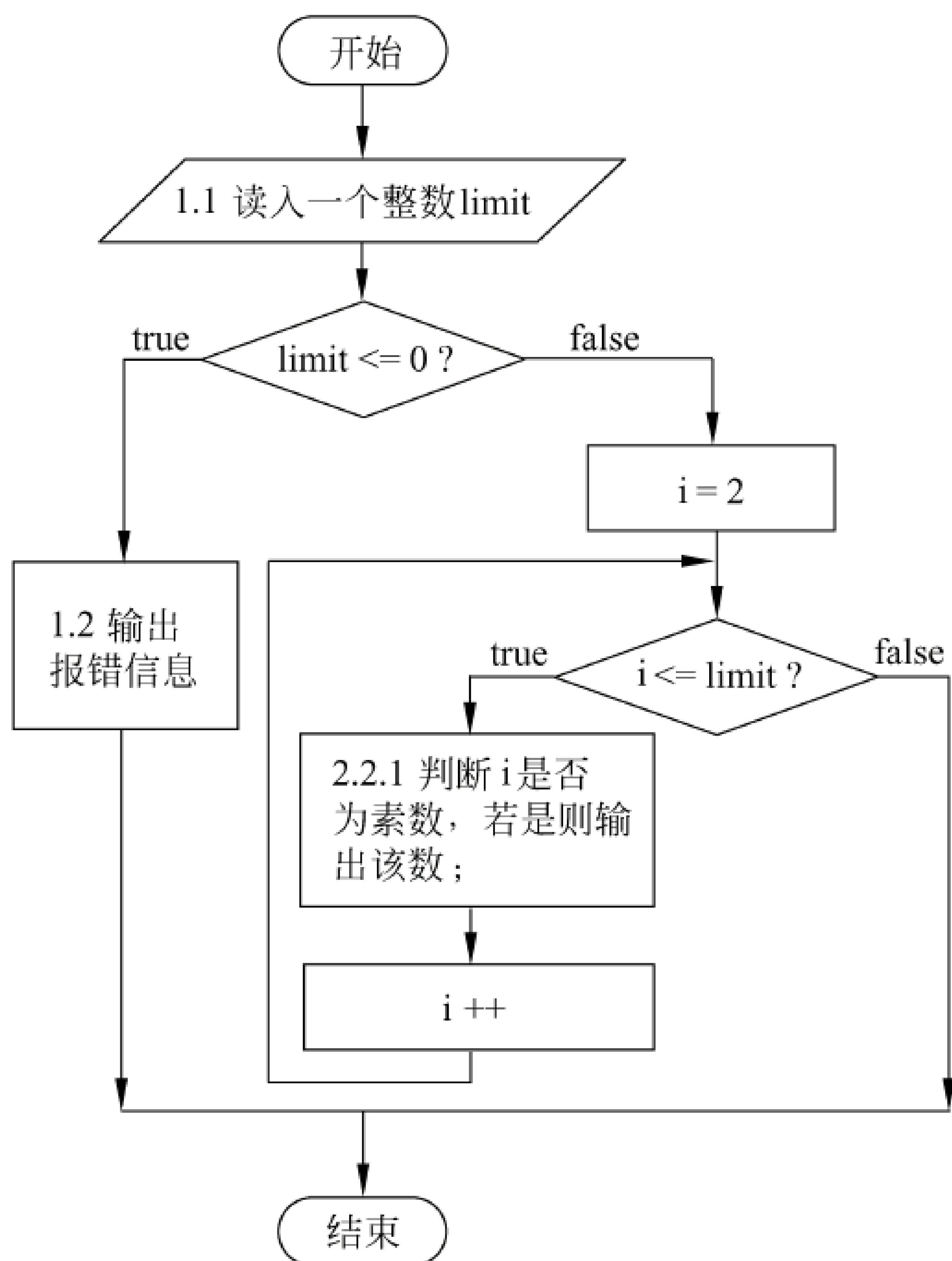


图4-16　对第2步“输出从2到该正整数之间的所有素数”求精后

继续对第2.2.1步"判断i是否为素数,若是则输出该数"进行细化。本次求精将明确判断素数的算法:

判断整数i是否是素数的简单方法是,看i是否能被2~i/2之间的某个整数整除,如果是,那么i就不是素数;否则是素数。

至此,通过对算法进行逐步求精,最后得到的算法如图4-17和图4-18所示。显然,算法求精到这种程度,已经能够很方便地变换成程序了。

文字描述:

1. 输入一个正整数;
   1.1 读入一个整数limit;
   1.2 判断limit是否小于等于0,若是则输出报错信息,结束程序;
2. 输出从2到该正整数之间的所有素数;
   2.1 循环控制变量i置2;
   2.2 判断i <= limit,
      2.2.1 若是,则判断i是否为素数,若是则输出该数;
         2.2.1.1 循环控制变量divisor置2;
         2.2.1.2 判断divisor <= i/2,
            2.2.1.2.1 若是,则判断i是否能被divisor整除(i % divisor==0),若能整除,则退出内层循环,转2.2.1.4;
            2.2.1.2.2 否则,转2.2.1.4;
         2.2.1.3 j自增,转2.2.1.2;
         2.2.1.4 若divisor > i/2,则i是素数,输出i,转2.3;
      2.2.2 否则,转3;
   2.3 i自增,转2.2;
3. 结束。

图4-17 对第2.2.1步"判断i是否为素数,若是则输出该数"求精后的算法文字描述

```
//ex4_16.cpp: 输入一个正整数,输出从 2到该正整数之间的所有素数
#include< iostream.h>

int main()
{
    int i, divisor, limit;
    //输入一个正整数
    //读入一个整数 limit
    cout<< "Please enter a positive integer number: ";
    cin>> limit;
    //判断 limit是否小于等于 0,若是则输出报错信息,结束程序
    if ( limit<=0 )
    {
        cout<< limit<< "is not a a positive integer number. Program exit.";
        return 0;
    }
    //输出从 2到 limit之间的所有素数
```

```
    for ( i=2; i<=limit; i++)
    {
        for ( divisor=2; divisor<=i / 2; divisor++)
            if ( i % divisor==0 )
                                    //判断 i是否能被 divisor整除 (i% divisor==0),若能整除,退出内层循环
                break;

        if ( divisor> i / 2)
            //若 divisor> i/2,则 i是素数,输出 i
            cout<< i<< " "

    }

    return 0;

}
```

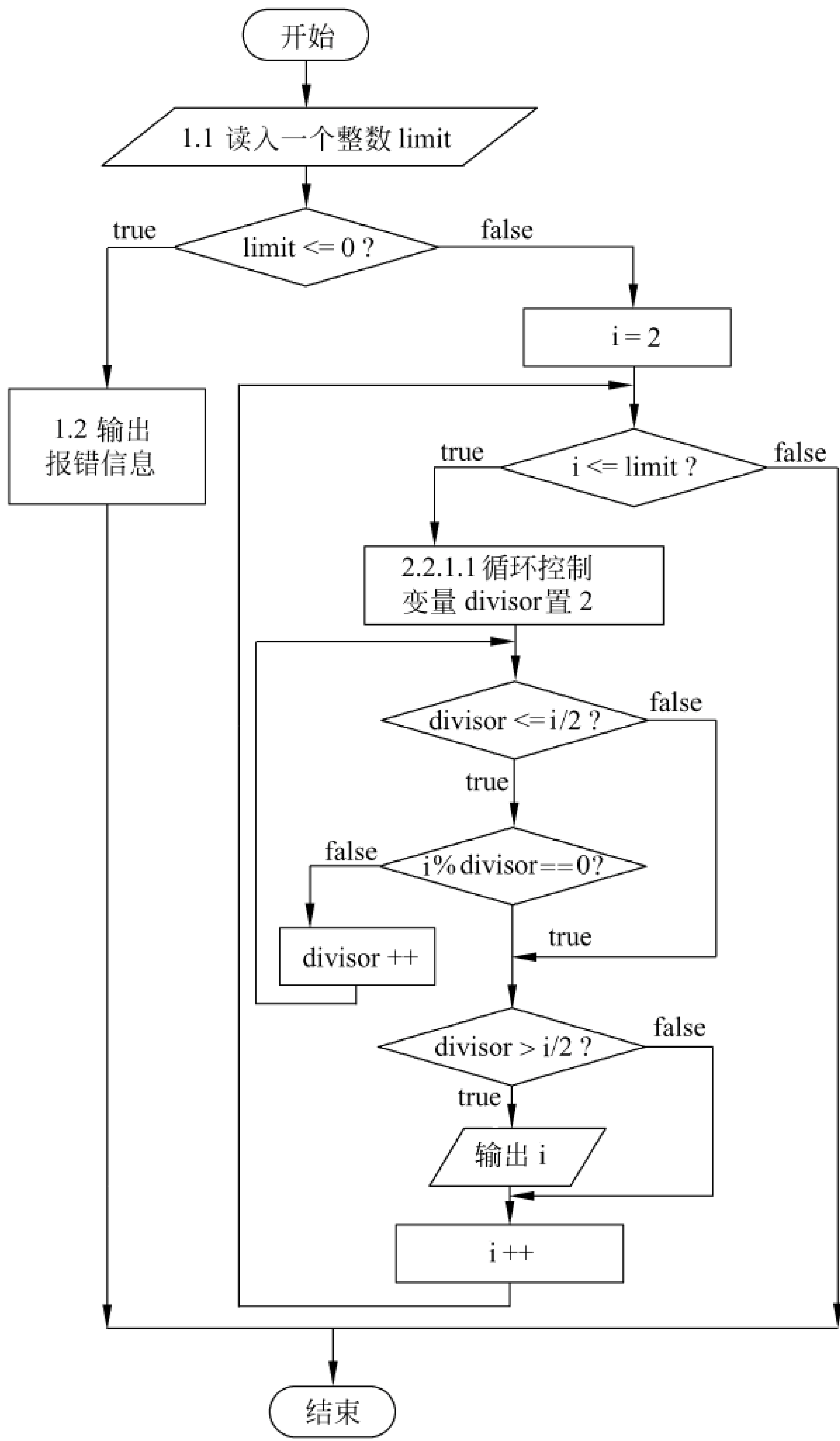


图 4-18　对第 2.2.1 步“判断 i是否为素数,若是则输出该数”求精后算法流程图

程序执行结果:

```
Please enter a positive integer number: 20
2 3 5 7 11 13 17 19
```

从这个例子可以看出,自顶向下、逐步求精的方法是一种行之有效的结构化程序设计方法。它使得人们能够从分析问题开始,对问题一步一步地深入剖析,随着不断地细化,离问题的解决也越来越近,最终得到一个能够正确执行的程序。

可能有些读者认为,问题本身并不复杂,有必要如此煞费苦心地通过逐步细化来进行算法设计吗?有些初学的程序设计者,甚至不经过算法设计,而直接就在机器上编写程序,以求获得快速的开发效率。其实“磨刀不误砍柴工”,良好的设计是编写正确、高质量程序的基础,尤其当问题复杂时,设计的意义更重要。解决实际问题时,在设计上花费的代价往往会明显超过用于编写代码的代价,无数经验表明,不经过精心设计,只能带来编程效率的下降和软件质量的低下,只要设计完善细致,程序编写就会变得简单和直接明了。

程序设计风格提示:采取逐步求精完成算法设计后,得到的算法应该直接成为程序的一部分:顶层算法描述就是程序功能的注解说明,较粗的算法步骤应该成为程序中的注解,较细的步骤则变换成程序代码。令初学者苦恼的变量命名也有了依据:应该尽量采用算法中出现的词作为程序相应变量的名字。可见,先进行设计,再编写程序,是养成良好程序设计风格的基础和保证。

4.7.4 再谈程序设计风格

第1章已讨论过程序设计风格问题,在此基础上,这里再针对结构化程序设计的特点进一步讨论C++的程序设计风格。程序员在编码时在源程序文件、语句结构、程序注释等方面要遵循一贯、标准的方法和规则。好的编码风格不仅能提高程序的质量、保证符合设计所要求的程序功能,而且还应该易读、易懂、易维护。下面补充一些好的程序设计风格的原则。

1. 清晰地书写代码

如果没有必要,尽量不要使用语言中生僻的特性,因为这些特性不易于理解和调试,应当采用最自然、最有利于保持良好结构的实现途径,使用大多数程序员都能理解的语言成分来书写代码,不易犯错且易于理解和维护。

2. 编写安全健壮的代码

进行程序设计时,往往将大部分注意力放在处理程序的正常流程上,总是假想用户会和程序员一样使用软件,然而用户总是会在与软件的交互过程中采取一些超出程序员设想的动作,如果程序设计时未考虑这些因素,就会导致软件的崩溃。因此,在设计时应当考虑用户所有可能的动作,特别是超出正常程序逻辑以外的异常情况。

3. 用简单的语句行

在C++语言中,一行可写多个语句。但调试是面向行的,如果一行程序过于复杂,将难于调试。因此,从调试的角度出发,每一个语句都应独自成行。

4. 适当地应用括号

用括号使书写清晰。表达式中的括号能明显地改善可读性。多使用括号还可减少对语法理解的歧义。由于读程序的人不一定能完全记住各种运算符的优先级和结合律,而且使

用多余的括号并不影响编译后的代码，因此，如果在容易引起误解的地方增加括号，可以提高可读性。

5. 灵活使用 C++ 的控制结构

用缩进表示循环和选择结构能使程序结构清晰、易读。

6. 充分利用 C++ 的特性

例如，可用输入输出流(iostreams)代替 stdio，以充分利用 C++ 编译器的特性。

7. 使用头文件

把所有的共享定义放在头文件中，以利于保持程序的一致性。

8. 初始化变量

在使用变量之前一定要把它们初始化。在初始化之前使用变量会产生错误。

9. 利用空行和空格

空行和空注释行能显著地改善可读性。例如，在程序中，说明部分和执行部分之间，一个逻辑相关联部分和另一部分开始之间都可以用空行显式地隔开。

水平方向加适当的空格可突出运算的优先性，避免发生运算错误，改善可读性。

例如：

```
(a<-17)&&!(b<=49)||c
```

写成

```
(a<-17 ) && ! ( b<=49 ) || c
```

在程序设计时应当尽可能遵循上述原则，保持良好的程序设计风格。

习　题　4

4.1　条件语句的格式如何？else 如何与 if 配对？

4.2　开关语句和格式如何？break 语句在开关语句中有何作用？

4.3　C++ 中提供的 3 种循环语句各有什么特点？

4.4　break 语句和 continue 语句在循环体内的作用有何不同？

4.5　编写程序，看看 exit()和 abort()函数有何不同？

4.6　下列程序打印出什么结果？

```
#include<iostream.h>
int main()
{
  int count=1;
  while (count<=10) {
     cout<<(count % 2 ?  "****" : "++++++++")<<endl;
    ++count;
  }
  return 0;
}
```

4.7　下列程序打印出什么结果？

```
#include<iostream.h>
int main()
{
  int row=10,  column;
  while (row>=1) {
     column=1;
     while (column<=10) {
       cout<<(row%2?"<":">");
       ++column;
     }
  --row;
  cout<<endl;
  }
  return 0;
}
```

4.8　下列程序的作用是什么?

```
#include<iostream.h>
int main()
{
    int x,y;
    cout<<"Enter two integers in the range 1-20: ";
    cin>>x>>y;
    for (int I=1;I<=y; I++) {
       for (int j=1; j<=x; j++)
         cout<<'@ ';
       cout<<endl;
    }
    return 0;
}
```

4.9　寻找下列各题的错误(注意,可能有多处错误)。

(1) 下列代码试图判断并输出整数 value 是奇数还是偶数:

```
switch (value%2) {
   case 0:
       cout<<"Even integer"<<endl;
   case 0:
       cout<<"odd integer"<<endl;
}
```

(2) 下列代码输出 19 到 1 之间的奇数:

```
for (x=19; x>=1; x+=2)
   cout<<x endl;
```

(3) 下列代码输出 2～100 之间(含 2 和 100)的偶数:

```
counter=2;
```

```
do {
  cout<< counter<< endl;
  counter+=2;
} while (counter< 100);
```

4.10 设计一个程序，输入实型变量 x 和 y，若 x>y，则输出 x−y；若 x<y，则输出 y−x。

4.11 输入一个不多于 5 位的正整数，要求：(1)求出它是几位数；(2)分别打印出每一位数字；(3)按逆序打印出各位数字，如原数为 321 则应输出 123。

4.12 编写一个程序，输入 10 个数，确定和打印其中最大数。

4.13 编写一个程序，读取 3 个 float 类型的非 0 值，并确定和打印其能否构成一个三角形的三条边。

4.14 编写一个程序，读取 3 个非 0 整数值，并确定和打印其能否构成一个三角形的三条边。

4.15 编写一个程序，打印整数 1～156 的二进制、八进制、十六进制和十进制对照表。

4.16 输入 4 个整数，要求按由大到小的顺序输出。

4.17 编写一个求解一元二次方程的根的程序。方程的系数由用户在运行程序时输入。

4.18 设计一个程序，将键盘上输入的百分制成绩转换成对应的五分制成绩并输出。若 90 分以上为 A，80～89 为 B，70～79 为 C，60～69 为 D，60 分以下为 E。要求分别用条件语句和开关语句完成。

4.19 求一个正整数的所有因子。例如，24 的因子是 1、2、3、4、6 和 12。

4.20 输入平面直角坐标系中一点的坐标值(x，y)，判断该点是在哪一个象限中或哪一条坐标轴上。

4.21 有一分数数列 2/1，3/2，5/3，8/5，13/，…。求出该数列前 20 项的和。

4.22 求出所有“水仙花数”。所谓“水仙花数”，是指一个 3 位数，其各位数字的立方和等于该数本身。例如，153 是“水仙花数”，即 $153=1^3+5^3+3^3$。

4.23 求出 1000 之内的“完数”。所谓“完数”就是指这个数等于它的因子之和。例如，6 是一个完数，即 6=1+2+3。

4.24 设计一个程序，按下列公式求出数列的前 20 项并输出。公式为：
当 n=0 时，y=0；
当 n=1 时，y=1；
当 n=2 时，y=2；
当 n>2 时，$y=y^{n-1}y^{n-2}+y^{n-3}$。

4.25 编写程序将华氏温度转换为摄氏温度。公式为：c=(F−32)×5/9。其中，c 为摄氏温度，F 为华氏温度。输入一华氏温度，转换为摄氏温度输出。

4.26 编写一个程序求一系列整数的和。假设第一个读取的整数指定后面要输入的数值的个数，程序在每个输入语句中只读取一个值。典型的输入序列如下：

```
5 100 200 300 400 500
```

4.27 编写一个程序打印如图 4-19 所示的图案。使用 for 循环分别打印每一个图案。所有星号(*)应在一条 cout<<'*'的语句中打印(使星号靠在一起)。提示：最后两个图案要求每一行以适当空格数开始。附加部分：将这 4 个问题的代码组合在一个程序中，利用嵌套 for 循环使 4 个图案并排打印。

4.28 两个乒乓球队进行比赛，每队各出 3 人。甲队为 A、B、C3 个人，乙队为 X、Y、Z 3 个人。已抽签决定比赛名单。有人向甲队队员打听比赛的名单，A 说他不和 X 比，C 说他和 X、Z 比，请编程序找出 3 对赛手的名单。

4.29 设用 100 元钱买 100 支笔，其中钢笔每支 3 元，圆珠笔每支 2 元，铅笔每支 0.5 元，问钢笔、圆珠笔和铅笔可以各买多少支(每种笔至少买 1 支)？
提示：设钢笔、圆珠笔和铅笔各买 i、j、k 支，则应有下列式子成立。

```
3* i+2* j+0.5* k=100
i+j+k=100
```

用穷举法将购买钢笔、圆珠笔的数量用二重循环遍历一遍。从中找出符合上述条件的购买方法。

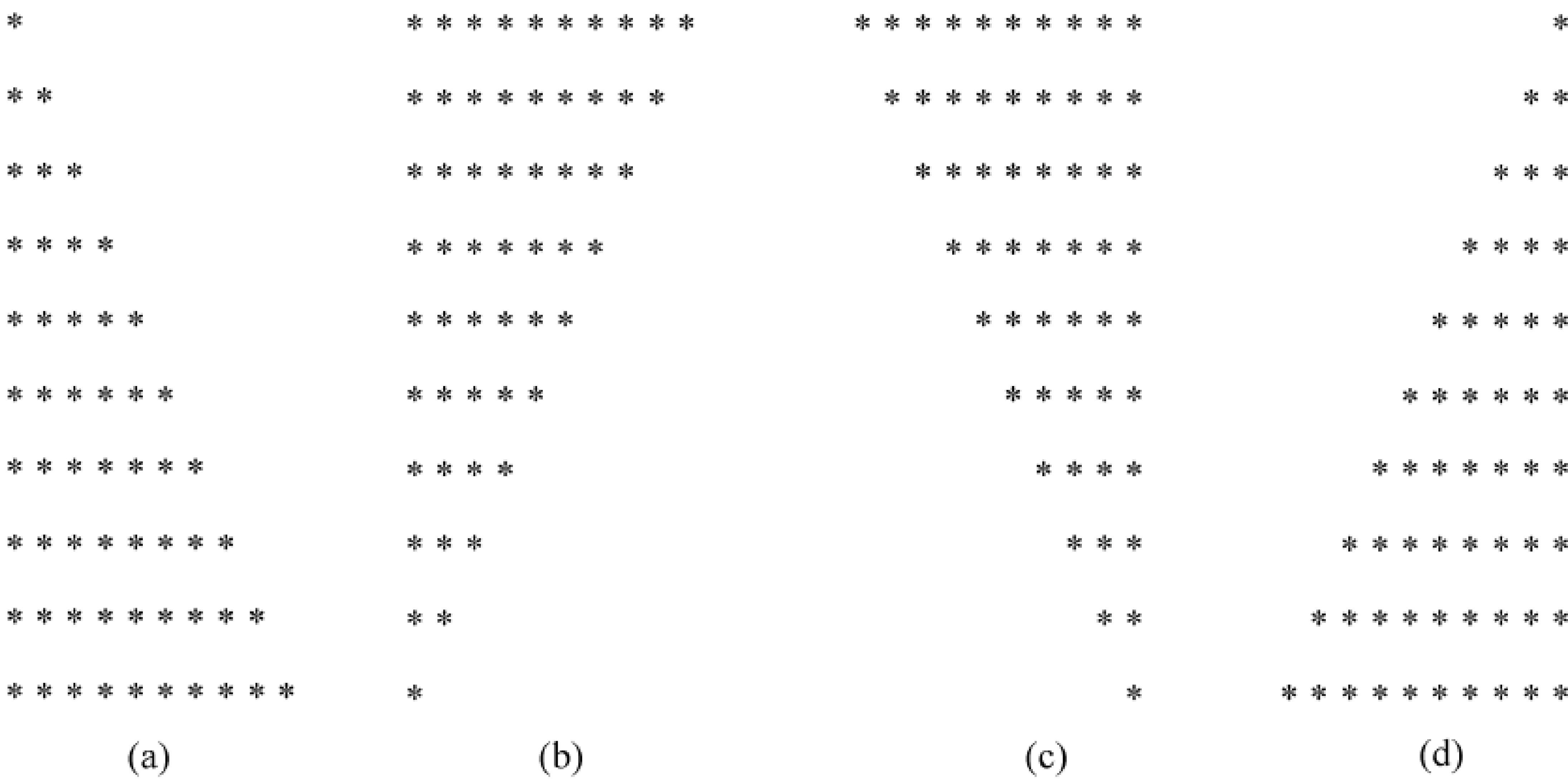


图 4-19 题 4.27 的图案

第 5 章 函　数

【学习内容】

本章将介绍 C++ 的函数机制，学习模块化程序设计思想。主要内容包括：

◆ 函数的定义和函数原型。

◆ 函数调用和参数传递机制。

◆ 函数重载。

◆ 存储类别和作用域。

◆ 递归函数设计和函数的递归调用。

◆ 预处理指令。

【学习目标】

◆ 掌握函数的定义和使用方法，理解函数原型、函数重载。

◆ 掌握传值和传引用两种参数传递的机制。

◆ 理解存储类别和作用域。

◆ 理解函数递归调用的执行过程。

◆ 能够熟练利用函数进行自顶向下、逐步求精的程序设计。

通过前面的学习，现在已经可以设计程序了，但能设计的程序都非常简单，真正有实际应用价值的程序都比较复杂。那么，如何设计复杂的程序？如何组织大型的软件呢？通过本章的学习，读者将了解函数机制，掌握通过函数实现程序的模块化的方法。

5.1 模块化程序设计

前面介绍过模块化的思想：解决一个复杂问题时，要把它分解成若干稍小的、简单的部分，每个部分对应一个模块进行问题求解。因此，要设计一个解决复杂问题的较大的系统，可以按子结构之间联系的疏密程度分解成较小的模块，每个模块完成一定子问题的求解，整个程序是由层次的逐级抽象的诸模块组成。在 C++ 中，每个模块可以是一个类或函数。

函数是 C++ 程序构成的基础，相当于其他高级语言中的子程序。C 程序就是由一个个函数组成的，由于 C++ 是 C 的超集，所以一个 C++ 程序也可以像传统的 C 程序一样，也由一个个函数组成。但是，采用面向对象的程序设计方法设计的 C++ 程序是由一个个类构成的，即便如此，任何一个 C++ 程序也至少包含一个函数——main()函数。函数也是类的方

法的实现手段。因此,在C++中,函数的作用有两个:一是按功能将一个复杂的大任务划分为若干小任务时,这些小任务可用函数来实现;二是在类中,用函数来定义类对象的方法,详见本书第10章及其之后的章节。一个C++程序无论有多么复杂、规模有多么大,程序的设计最终都落实到一个个函数模块的设计上,通过对函数模块的调用实现程序的特定功能。

C++中的函数包括两类:预定义函数和用户自定义函数。C++提供了极为丰富的库函数,这些库函数是预定义好的,程序员可以在自己的程序中直接使用。当然,还需要将包含欲使用的函数声明的头文件包括进来。另外,C++还允许用户根据需要建立自定义函数,把自己的算法编成一个个相对独立的函数模块,然后像调用库函数一样来调用这些函数。C++程序一般都由程序员编写的各种新函数与C++标准库中提供的预定义函数组合而成。

C++继承了ANSI C语言在结构化程序设计上的优点,通过采用函数模块式的结构,C++语言易于实现结构化程序设计,使程序的层次结构清晰,便于程序的编写、阅读和调试。其实,结构化程序设计的方法也是面向对象程序设计的基础,因为,在面向对象程序设计中,一个类的方法的实现,最后还是要使用结构化程序设计手段。本章重点学习如何用C++的函数机制实现结构化和模块化的程序设计思想。

使用C++进行模块化程序设计时,一个C++完整的程序可包含多个文件,一个文件中可包含多个函数。此外,文件是通常意义上的"模块",它也是编译单位。因此,当某一文件内容改变时,只需单独编译此文件,然后将它们与其他编译过的文件进行链接,就可生成可执行程序。

5.2 预定义函数的使用

C++提供了丰富的预定义函数,也称为库函数。这些库函数的声明都放在相应的头文件中,库函数的声明说明了库函数的名字、参数个数及类型、函数返回结果的类型等信息。使用这些库函数时,首先要将相应的头文件包括进来,然后就可直接使用库函数了。例如,数学库函数中的函数能完成一般数学运算,其声明包含在math.h中,使用这些函数时应加下面的代码:

```
#include<math.h>
```

在程序中要调用库函数时,遵循下面的调用格式:

<函数名>(<参数列表>)

括号内是提供给被调用函数的参数,称为实在参数,简称实参。如果有多个参数,参数之间用逗号分隔,提供的参数可以是常量、变量和表达式。要求调用函数时提供的参数在数目、类型上必须与函数的声明一致。函数可以有一个返回结果,所以函数调用本身可以构成表达式的一部分。

函数调用的一般过程是,先计算实参表达式的值,将计算的结果交给被调用函数(通过后面介绍的参数传递机制),接着执行被调用函数的代码,直至执行到返回语句return或执

行到函数末尾,程序控制返回到调用函数的调用处,继续执行。

例如,求平方根函数 sqrt 的声明在 math.h 中,该函数接受一个 double 类型的参数,函数的返回结果也是 double 类型,假设 main 函数中有下面的代码:

```
printf("%.2f", sqrt(900.0));
```

执行过程是:以 double 类型的 900.0 为参数调用 sqrt 函数,该函数计算参数 900.0 的平方根,返回结果 30.0;然后,以%.2f 和 30.0 为参数调用 printf 函数,实现格式化输出。这里 printf 也是预定义库函数,其声明在 stdio.h 中。

除了预定义的库函数外,程序员还可根据实际需要、按照模块化原则设计自己的函数,然后通过调用这些函数完成相应的功能。下面介绍用户自定义函数的定义和使用。

5.3　函数定义与函数原型

函数是一个命名的程序代码块,是程序完成其操作的功能单位。在程序设计中,有许多算法是通用的,例如求一个数的平方根,解一元二次方程。如果将这些算法定义为一个函数,就可以在程序中需要这些算法的地方直接使用它们(而不必再作定义)。C++ 的库函数已经提供了丰富的功能,但是,进行程序设计时,程序员往往还需要根据具体问题的需求设计自己的函数模块,为此,应该先定义一个函数,然后就可以像使用预定义函数一样使用自定义函数。定义一个函数就是编写完成该函数功能的程序块。

5.3.1　函数定义

1. 函数定义格式

函数定义的一般格式为:

```
<返回值类型><函数名>(<参数列表>)
<函数体>
```

其中:

<返回值类型>说明函数返回值的数据类型,简称为函数的返回类型。它可以是任一基本数据类型或用户自定义的数据类型;如果无返回值,则用关键字 void 说明。默认的返回类型是 int,即若未指定返回类型,则返回类型是 int(注意不是 void)。

<函数名>是程序员为该函数指定的名字,函数名应遵守标识符命名的规定。

<参数列表>指明参数的个数、名称和类型,函数定义中的参数称为形式参数,简称形参。有多个形式参数时,用逗号分隔。如果函数没有形参,参数列表为空,函数名后面的圆括号不能少,这时也可在括号中加上关键字 void,表示这是一个无参函数。

<返回值类型>、<函数名>及<参数列表>构成了函数头。

<函数体>描述函数的功能,即函数所完成的具体操作,它由一系列说明语句和执行语句组成。函数体实际上是一个复合语句,花括号不能少,它指明函数体的开始和结束。函数执行时,如同执行一个复合语句一样,顺序执行函数体,直到遇到 return 语句或者遇到表示函数体结束的右花括号为止,函数执行完毕后,返回调用程序继续执行。

例如:

```
int max(int a, int b, int c)
{
    int m;
    m= (a>b) ?a : b;
    return (m>c) ?m : c;
}
```

程序定义了一个名为max的函数,该函数有3个形式参数a、b和c,类型均为int,返回值也是int类型。函数体由一个变量说明语句、一个赋值语句和一个返回语句构成。函数的功能是求a、b和c中最大的值。

兼容性提示:和C++不同,在C语言中,空的参数列表表示函数可以接受任意多个参数,参数列表必须为void才表示无参函数,所以对于无参函数最好在参数列表中写上void。

所有函数的定义是并列的、平行的,在一个函数定义内部不允许定义另外一个函数。但可以对别的函数进行调用或作引用说明。

函数定义中声明的所有变量都是局部变量(local variable),只在声明语句所处的程序块中有效。大多数函数都有一组参数,函数定义时指明了每个形式参数的名字、类型,形式参数与函数调用时提供的实在参数一一对应,函数执行时,形式参数可以视为实在参数的代表。参数机制提供了函数之间沟通信息的方式。函数的形式参数也可视为局部变量,在函数体范围内有效。

2. 函数的返回值

根据返回值的情况,函数可以分为两类:无返回值函数和有返回值的函数。

1) 无返回值函数

函数无返回值时,返回类型必须用关键字void说明。这类函数的函数体内无return语句或者return语句中无表达式。

如果函数体中包含不带表达式的return语句,当函数体执行到这样的return语句时,即返回到调用程序;如果函数体无return语句,当执行完函数体最后的语句,遇到表示函数体结束的右花括号时,返回到调用程序。

2) 有返回值的函数

函数返回值的类型如果不是void,表示该函数有返回值,此时函数体必须包含带表达式的return语句。表达式的值将返回给调用程序,该值具有函数说明的返回值类型。函数返回值的类型可以是算术类型,也可以是后面将介绍的指针、数组名、引用以及用户定义的数据类型(如结构、联合或类等)。

不论函数有无返回值,函数体可有多个return语句。函数调用时,只要遇到一个return语句,就将忽略函数体中其余的代码,立刻返回到调用程序。例如,上面定义的函数max计算并返回3个整数中的最大值。

函数一经定义,就可在程序中多次地使用。函数的使用是通过函数调用实现的。

5.3.2 函数原型

C++的重要特性之一是函数原型(function prototype)。函数原型告诉编译器该函数的

函数名字、函数返回的数据类型、函数要接收的参数个数、参数类型和参数顺序。编译器通过函数原型来验证函数调用的正确性。

函数原型的常用形式如下：

```
<返回值类型><函数名>(<参数列表>);
```

函数原型看上去只是比函数头多了一个分号，实际上，函数原型的参数列表中形参的名称可以省略。因为函数原型的意义在于验证函数调用时实在参数和形式参数的一致性，而不用关心形式参数的名字，因而只要指明参数的个数、类型及顺序就足够了。

例如，函数 max 的原型为

```
int  max(int , int , int);
```

这个原型表示函数 max 具有 3 个 int 参数，返回 int 类型结果。注意这个函数原型与 max 函数定义的首部相似，只是不包括参数名 a、b 和 c。

函数原型中包括的函数名和参数类型部分称为函数签名(function signature)或签名(signature)。函数签名不包括函数返回类型。

程序设计风格提示：如果有几个函数在同一个文件中，一般把主函数放在前，被调用函数的定义放在后，并将其原型放在主函数前进行声明，以符合“先声明后使用”的原则。

5.4　函数的调用

5.4.1　函数调用的概念

在 C++ 程序中，除了 main()函数以外，任何一个函数都不能独立地在一个程序中存在，都是通过 main()函数直接或间接地调用引发执行的。调用一个函数就是执行该函数的函数体的过程。

函数调用的一般形式为：

```
<函数名>(<实在参数列表>)
```

<函数名>是用户自定义的函数名或者系统预定义的标准函数名。<实在参数列表>是一个表达式列表。实在参数与形参表中的形式参数要顺序一致、类型匹配、个数相同(可变参数除外)。如果被调用函数是无参函数，它被调用时就没有实参，但圆括号必须有。如果有形式参数，相应的实在参数应该是类型一致的表达式，包括常量、变量或后面介绍的数组元素等。

C++ 的函数调用可以出现在表达式中，这种情况下要求被调用的函数应该有返回值。对于没有返回值的函数，也可以单独构成函数调用语句，函数调用语句就是在函数调用后面加上分号，即：

```
<函数名>(<实在参数列表>);
```

函数调用时，先计算每个实参表达式的值，将实参的值赋给被调用函数对应的形式参

数,然后再执行被调用函数的函数体。函数体执行时,顺序执行各语句,直到遇到 return 语句或者遇到表示函数体结束的右花括号为止,被调用的函数执行完毕,返回调用程序继续执行。如果被调用函数有返回值,系统从被调用函数返回到调用程序时,会将 return 语句中的表达式的值返回给调用程序,调用程序可以继续使用该值进行后续运算;如果被调用函数没有返回值,则被调用的函数执行完毕时,直接返回到调用程序,继续执行。图 5-1 给出了调用 max 函数的示意图。

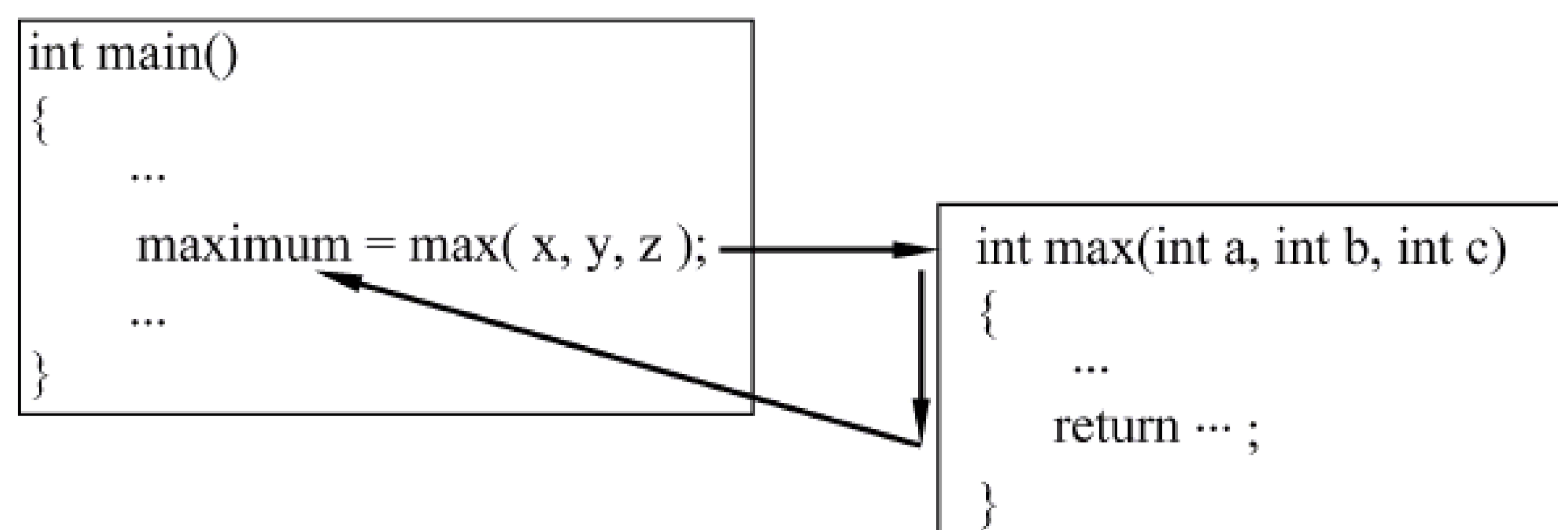


图 5-1　函数调用示意图

如果实在参数与形式参数的类型不同,则可能需要把实参的类型转换成对应形参的类型。例如,数学库函数 sqrt 可以用整型参数调用,虽然 math. h 中的函数原型指定为 double 参数,但函数仍然可以顺利工作。下列语句

```
cout<< sqrt(4);
```

先隐式地将 int 类型的 4 强制转换成 double 类型,然后再用 double 类型的 4.0 调用 sqrt,所以能正确地求出 sqrt(4)的值为 double 类型的 2.0。一般来说,与函数原型中参数类型不完全相符的参数值将被隐式转换为正确类型之后,才能进行函数调用。

将数值转换为较低类型时可能导致数据丢失精度。为了避免在编写调用代码时出现这类错误,C++ 不会自动将精度较高类型的实在参数自动转换成精度较低类型的形式参数。如果确实要将精度较高类型的实在参数值传递给精度较低类型的形式参数,只能先显式地将该值赋给较低类型的变量,再用该变量作为实在参数来调用函数,或者在实参表达式中使用强制类型转换运算符实现类型转换。这种要求可以看成是保证程序安全性的一种措施。例如,对于函数 max,其原型为:

```
int max(int, int, int);
```

如果变量 x、y 和 z 是 double 类型的,下面第(1)、(2)种对 max 的调用方式是正确的,而第(3)种调用方式将导致警告性错误。

```
(1) max((int)x, (int)y, (int)z);
(2) int nx= (int)x;
    int ny= (int)y;
    int nz= (int)z;
    max(nx, ny, nz);
(3) max(x, y, z);
```

【例 5-1】　用自定义函数计算并返回 3 个整数的最大值,并输出结果。

```
//ex5_1.cpp: 用自定义函数计算并返回 3个整数的最大值,并输出结果
```

```
#include<iostream.h>
int max(int, int, int);
int main()
{
    int x, y, z;
    int maximum;
    cout<<"Please enter three integers: ";
    cin>>x>>y>>z;
    maximum=max(x, y, z);
    cout<<"The maximum integer is: "<<maximum<<endl;
    return 0;
}
int max(int a, int b, int c)
{
    int m;
    m=(a>b)?a:b;
    return (m>c)?m:c;
}
```

程序执行结果：

```
Please enter three integers: 12 34 23
The maximum integer is: 34
```

【例5-2】 用自定义函数求两个整数的最大公因子，并编写程序调用该函数输出结果。

设计思想：本题解法可分为3步：输入数据，求最大公因子，输出结果。核心问题是如何设计一个函数求两个整数的最大公因子。可以采取下面的辗转相除方法。

对于a和b两个整数，假设a>b，如果a不能被b整除(余数记为remainder)，则反复执行下面的步骤，直到a能被b整除：

```
a=b;
b=remainder;
```

程序实现如下：

```
//ex5_2.cpp: 用自定义函数求两个整数的最大公因子
#include<iostream.h>
int gcd(int a, int b)
{
    int temp;
    int remainder;
    //如有必要,通过交换使得 a>=b
    if (a<b)
    {
        temp=a;
```

```
        a=b;
        b=temp;
    }
    remainder=a % b;
    while (remainder != 0)
    {
        a =b;
        b=remainder;
        remainder=a % b;
    }
    return b;
}
int main()
{
    int x, y;
    int fac;
    cout<<"Please enter two integers: ";
    cin>>x>>y;
    fac=gcd(x, y);
     cout<<"The greatest common divisor of ";
    cout<<x<<" and "<<y<<" is: "<<fac<<endl;
    return 0;
}
```

程序执行结果：

```
Please enter two integers: 120 64
The greatest common divisor of 120 and 64 is: 8
```

5.4.2 参数传递

在调用带参数的函数时，需要进行参数传递。参数传递提供了调用函数和被调用函数交换信息的渠道。C++中的参数传递一般采用的是传值(call by value)方式，即传入的只是参数的值的复制，而不是参数的地址，因此被调用函数执行时，只能访问形式参数对应的内存单元(称为形式单元，单元中存放了实在参数的值)，而不能直接访问实在参数对应的单元，因而无法直接改变实在参数的值。下面的swap函数试图交换两参数的值。

```
#include<iostream.h>
void swap(int x,int y)
{
    int temp;
    temp=x;
    x=y;
    y=temp;
```

```
}
int main()
{
    int x=10, y=20;
     swap(x, y);
     cout<<"x:"<<x<<"y:"<<y<<endl;
    return 0;
}
```

程序输出：

```
x:10   y:20
```

运行结果显示，x 与 y 的值并没被交换，这是因为 main()函数在执行 swap(x, y)时，是把实在参数 x 和 y 的值传给了 swap 中形式参数 x 和 y(注意，实参 x、y 与形参 x、y 是不同的变量，都有各自不同的存储单元)，swap 函数中交换的也只是形参 x 和 y 这两个局部变量的值，修改的是形式单元中的值，当 swap 函数执行完后，控制返回 main()函数的调用语句处，继续向下执行，此时形参 x 和 y 这两个局部变量也被撤销。可见，在整个 swap 函数执行过程中，形式参数值的改变丝毫也没影响调用函数中实在参数 x 和 y 的值。

从上面的程序可以看出，参数传递允许信息从调用函数流向被调用函数，而被调用函数想要将数据传给调用函数，只能通过返回结果这一途径。在许多情况下，这一限制为程序设计带来了不便，因为它使得从被调用函数向调用函数传递的信息非常有限。为了弥补这一不足，使被调用的函数能够访问调用它的函数的变量，C++ 还提供了传引用(call by reference)这一传递参数的方式。

如果采用传引用方式进行参数传递，要求实在参数只能是变量。当调用一个函数时，是把对实参变量的引用传给形式参数，即将实参变量的地址存放到对应的形参的形式单元中。当程序控制转入被调用函数后，在执行函数体过程中，对形式参数的任何引用或赋值都被处理成对相应形式单元的间接访问，即根据形式单元中存放的实参变量的地址找到实参单元，对形式参数的引用或赋值被处理成对该实参单元的访问。当被调用函数工作完毕返回时，形式单元所指的实参单元就持有了所期望的值。可见传引用的参数传递方式不仅可以将数据从调用函数传递给被调用函数，也可以通过形式参数访问调用函数中的实参变量，实现信息从被调用函数传向调用函数。

为了区分参数不同的传递方式，C++ 规定，按照传引用方式传递的参数，在说明形式参数时，参数名前要加上 & 符号，如果没有该符号，说明该参数按照传值方式进行传递。同一函数中，允许有的参数传引用，有的参数传值。

例如，为了实现两个整数的交换，上面的 swap 函数应该采取传引用方式。

【例 5-3】 用自定义函数实现两个整数的交换。

```
//ex5_3.cpp: 用自定义函数实现两个整数的交换
#include<iostream.h>
void swap(int &, int &);
int main()
{
```

```
        int x=10, y=20;
        swap(x, y);
        cout<<"x:"<<x<<"    y:"<<y<<endl;
        return 0;
}
void swap(int & a,int & b)
{
        int temp;
        temp=a;
        a=b;
        b=temp;
}
```

程序输出：

```
x:20   y:10
```

可见，采取传引用方式进行参数传递，形、实参结合时的语义与传值方式的调用是不同的，可以将形式参数视为对应的实参变量的别名，对形式参数的任何访问(包括赋值)，实际上是对实参变量的访问，因而可以实现对调用函数中实参变量的修改。

除了传引用方式以外，还可以通过传递指针参数来实现被调用函数访问调用它的函数的变量。

软件工程提示： 传值和传引用这两种方法各有优缺点。使用传值方式，可通过函数的返回值显式地将数据交给调用程序，从而改变调用程序中变量的值，但它不能返回多个值，返回值的类型也受到限制。采取传引用方式，可将更多的数据从被调用函数传向调用函数，但它不是显式地修改调用程序中的变量，损害模块的独立性，容易出错。

5.4.3 默认参数

为了方便程序设计，在函数调用时可能要为某些参数传递特定的值。程序员可以将这些参数指定为默认参数，并提供这些参数的默认值。当函数调用时，如果没有为默认参数提供相应的实参，系统将把预设的默认参数值传递给被调用函数的形参。

默认参数应在函数名第一次出现时指定，通常是在函数原型中，直接在参数说明中增加一个赋值符号和默认值，默认值可以是常量、全局变量或函数调用。

默认参数必须是函数参数表中最右边(尾部)的参数。调用具有两个或多个默认参数的函数时，如果省略的参数不是参数表中最右边的参数，那么该参数右边的所有参数(这些参数必定是默认参数)也应省略。默认参数也可用于即将介绍的内联函数中。

【例 5-4】 使用默认参数计算正方体的容积。

```
//ex5_4.cpp: 使用默认参数计算正方体的容积
#include<iostream.h>
//函数原型,指定默认参数及默认值
int cubeVolume( int length=1, int width=1, int height=1 );
int main()
```

```
{
   cout<< "The default cube volume is: "<< cubeVolume()
        << endl;
   cout<< "The volume of a cube with length 10,"
        << "default width 1 and default height 1 is: "<< cubeVolume( 10 )
        << endl;
   cout<< "The volume of a cube with length 10,"
        << "width 5 and default height 1 is: "<< cubeVolume( 10, 5 )
        << endl;
   cout<< "The volume of a cube with length 10,"
        << "width 5 and height 2 is: "<< cubeVolume( 10, 5, 2 )
        << endl;
   return 0;
}
//计算正方体的容积
int cubeVolume( int length, int width, int height )
{
   return length * width * height;
}
```

程序执行结果：

```
The default cube volume is: 1
The volume of a cube with length 10,default width 1 and default height 1 is: 10
The volume of a cube with length 10,width 5 and default height 1 is: 50
The volume of a cube with length 10,width 5 and height 2 is: 100
```

5.4.4　内联函数

从软件工程角度看，将程序用一组函数来实现有很多好处，如提高程序的模块化程度，但函数的调用却会增加程序执行时的开销，包括参数传递、转入被调用函数和返回调用函数，都带来了额外的开销。为此，C++ 提供了内联函数（inline function）机制，可以减少函数调用的开销。

声明一个函数是内联函数，只要在函数定义中函数返回值类型之前增加 inline 限定符即可。例如：

```
inline double cube(const double s) {
    return s * s * s;
}
```

inline 指示编译器在所有调用这个函数的地方将其实际的函数代码复制到程序中以避免函数调用。用一个函数代码的拷贝替换函数调用称作内联扩展或内联。其代价是会产生函数代码的多个副本并分别插入到程序中的每一个调用该函数的位置上（从而导致程序代码更多），而不是只有一个函数副本（每次调用函数时将控制传入函数中）。这一方式减小了

函数调用带来的开销,但使得程序代码的规模增大,所以对于较小函数更适宜采取内联方式。

内联函数的定义(即声明以及整个函数代码)必须出现在每一调用该函数的源文件之中。否则,编译器会因无法访问其函数代码而不能实现内联。可把该函数的定义放在源文件的头文件中,这样每一调用模块都可使用该定义的同一拷贝。和非内联函数不同,如果内联函数有了更改,调用它的所有源程序必须重新编译。

C++标准并未强制规定编译器一定要对具有 inline 限定符的函数进行内联扩展,这依赖于编译器的具体实现。例如,在许多编译器中,递归函数和用指针调用的函数即使用了 inline 指令,也不会被内联扩展。但需要指出,是否内联扩展并不会改变程序的语义,它只会影响到目标代码的执行速度和规模大小。一般情况下,内联扩展后速度会增快、代码将增多。因此,如果定义了一个小函数,整个代码调用它的地方相对来说又很少,则应该使用 inline 指令,尤其是在循环中重复调用此函数时。

【例 5-5】 使用内联函数计算 1~10 的所有整数的平方。

```
//ex5_5.cpp: 使用内联函数计算 1至 10的所有整数的平方
#include<iostream.h>

inline int square( int x ) { return x* x; }

int main()
{

    for (int i=1; i<=10; i++)
        cout<<"The square of integer "<<i<<" is "
            <<square(i)<<endl;

  return 0;
}
```

程序执行结果:

```
The square of integer 1 is 1
The square of integer 2 is 4
The square of integer 3 is 9
The square of integer 4 is 16
The square of integer 5 is 25
The square of integer 6 is 36
The square of integer 7 is 49
The square of integer 8 is 64
The square of integer 9 is 81
The square of integer 10 is 100
```

5.5 函数重载

在 C++中允许定义多个同名函数来表示类似的操作,只要这些函数有不同参数(参数的个数、类型或顺序不同),这种功能称为函数重载(function overloading)。调用重载函数

时,C++编译器通过检查调用中的实在参数个数、类型和顺序来确定相应的被调用函数。函数重载一方面能够减少函数名的使用,另一方面合理使用函数重载也能提高程序的可读性。例如,如果若干个函数是对不同的数据进行类似的操作,那么这些操作可以实现成一组重载的同名函数。因此,函数重载常用于设计多个功能类似而处理对象的类型不同的同名函数。

例如,C语言中有多个标准函数能够计算数值参数的绝对值,由于C语言中要求函数命名具有唯一性,所以需要为这些功能类似的函数取不同的名字。例如:

```
int iabs(int i);
long labs(long l);
double fabs(double d);
```

这几个函数的功能类似,但是使用3个不同的函数名,看上去就有些累赘,如果把3个函数定义为相同的名字,只是参数的类型不同,例如:

```
int abs(int i);
long abs(long l);
double abs(double d);
```

而在函数调用时,采用下面统一的调用形式:

```
abs(-10);                  //调用 int sbs(int)
abs(-1000000);             //调用 long abs(long)
abs(-3.1415926);           //调用 double abs(double)
```

既直观又简便。程序员不必根据参数的类型去判断到底应调用 iabs、labs,还是该调用 fabs,现在只要知道 abs 函数能够返回输入参数的绝对值就可以了。

C++之所以能处理重载函数得益于函数原型这一概念。函数原型不仅指出了函数名,也指出了函数的参数的数目、类型和顺序。在调用函数时,即使有两个函数的名字相同,只要参数的数目、类型或顺序不同,编译器就能够根据实在参数的数目、类型和顺序,找到唯一一个与之匹配的一个函数,这一过程称为绑定(binding)。例如,abs(-10)是调用的 int abs(int)而不是 long abs(long)或者 double abs(double),就因为其参数类型为 int。类似地,abs(-1000000)调用的是 abs(long)而不是 abs(int),就因为其参数类型为 long,而 abs(-3.1415926)参数类型为 double,因而它调用 double abs(double)。

【例 5-6】 使用重载函数计算 char 类型、int 类型以及 double 类型值的平方。

```
//ex5_6:使用重载函数计算 char类型、int类型以及 double类型值的平方
#include<iostream.h>

int square( char );
long square( int );
double square( double );

int main()
{
    char c=7;
    int i=1000;
```

```
    double d=3.1415926;
    cout<<"The square of char integer "<<(int)c<<" is "
        <<square( c )<<endl;
    cout<<"The square of integer "<<i<<" is "
        <<square( i )<<endl;
    cout<<"The square of double "<<d<<" is "
        <<square( d )<<endl;
    return 0;
}
int square( char x ) { return x* x; }
long square( int x ) { return x* x; }
double square( double x ) { return x* x; }
```

程序执行结果:

```
The square of char integer 7 is 49
The square of integer 1000 is 1000000
The square of double 3.14159 is 9.8696
```

5.6 存储类别

C++中的变量除了具有名字、类型和值以外,还有其他丰富的属性,如存储类别、作用域和链接特性等。本节将介绍变量的存储特性,下一节将介绍作用域。C++根据变量的存储特性来确定其生存期。所谓生存期,对于变量来说,表示该变量的存在的时间范围。而程序中一个标识符有效的代码范围称为其作用域。

5.6.1 变量的存储特性

2.3.2节中介绍变量声明的语法时曾指出,声明变量要指明变量的存储类别(storage class)。存储类别确定了变量占用内存空间的时间。在C++中,有4个存储类别描述符(storage class specifier)可以描述具有不同存储特性的变量,见表5-1。

表5-1 存储类别描述符及其意义

存储类别描述符	意 义	存储类别描述符	意 义
auto	自动变量	extern	外部变量
register	寄存器变量	static	静态变量

这4个存储类型说明符可以分为两大类:自动存储类别(automatic storage class)和静态存储类别(static storage class)。自动变量和寄存器变量属于自动存储类别的变量,外部变量和静态变量属于静态存储类别的变量,即关键字auto和register用来声明自动存储类别的变量,关键字static和extern则用来声明静态存储类别的变量和函数标识符。auto是默认的存储类别。在进行变量声明时,对一个变量不仅应说明其数据类型,还应说明其存储

类别。例如：

```
static int a,b;              //说明 a,b为静态整型变量
auto char c1,c2;             //说明 c1,c2为自动字符变量
extern int x,y;              //说明 x,y为外部整型变量
```

下面分别介绍以上4种存储类别。

5.6.2　自动变量

自动变量用auto声明。这种存储类型是C++程序中使用最广泛的一种类型。C++规定，函数内凡未加存储类别描述符的局部变量和参数均为自动变量。也就是说，自动变量可省去描述符auto。例如，下面某函数内的声明：

```
int i,j,k;
char c;
```

等价于：

```
auto int i,j,k;
auto char c;
```

自动变量具有以下特点。

（1）自动局部变量仅在定义该变量的函数体或程序块内有效，即在函数中定义的自动变量，只在该函数内有效，在复合语句中定义的自动变量只在该复合语句中有效。

（2）自动局部变量采取的是动态存储方式，只有当定义该变量的函数被调用时，才给它分配存储单元，开始它的生存期；当函数调用结束，就要释放该存储单元，生存期结束。因此函数调用结束之后，自动变量的值不能保留。在复合语句中定义的自动变量，在进入复合语句时建立该变量、分配存储单元，退出复合语句时，撤销该变量、释放存储单元，这之后该自动变量不能再使用，否则将引发错误。总之，自动变量的生存期开始于所在语句块的开始执行的时刻，终止于所在语句块的结束的时刻。

*5.6.3　寄存器变量

程序中大部分变量都存放在存储器内，而在计算机中，对数据进行计算和处理只能在寄存器中进行，因而当对一个变量频繁读写时，势必要反复访问存储器，将数据在存储器和寄存器之间频繁传送，花费大量的存取时间。为了在程序设计语言这一层次上支持效率优化，C++提供了另一种变量，即寄存器变量。这种变量被分配在CPU的寄存器中，使用时，不需要访问内存，直接从寄存器中读写，这样可以提高程序的执行效率。寄存器变量的描述符是register。对于循环次数较多的循环控制变量以及循环体内反复使用的变量均可定义为寄存器变量。

例如，求s＝1＋2＋3＋…＋200的程序段可以设计如下：

```
register i, s=0;
for(i=1;i<=200; i++)
s=s+i;
```

本程序循环200次,i和s都将频繁使用,因此可将它们定义为寄存器变量。

注意:只有局部自动变量和形式参数才可以定义为寄存器变量。因为寄存器变量除了是在寄存器中分配,其他性质与自动变量相同,都属于动态存储方式,凡需要采用静态存储方式的变量不能定义为寄存器变量。此外,将变量声明为寄存器变量,并不能保证编译器一定将该变量分配在寄存器中。

*5.6.4 外部变量

外部变量的存储类别描述符为extern。外部变量总是和全局变量联系在一起。所谓全局变量,是指那些在函数之外声明的变量,这些变量在整个文件范围内都有效。默认情况下,全局变量和函数名具有extern属性。外部变量的特点如下。

(1) 外部变量采取静态存储方式,即程序开始执行时就建立外部变量,直到整个程序结束(extern也可用来限定函数名,说明该函数名从程序开始执行时就可用)。

(2) 当一个源程序由若干个源文件组成时,在一个源文件中声明的外部变量需要在某个源文件中定义为全局变量。例如,下面源程序由源文件F1.cpp和F2.cpp组成。

```
//F1.cpp:
int a,b;                    /* 定义变量 a,b* /
char c;                     /* 定义变量 c* /
main()
{… }
//F2.cpp:
extern int a,b;             /* 声明 a和 b是外部整型变量,在 F1.cpp文件中定义 * /
extern char c;              /* 声明 c是外部字符型变量,在 F1.cpp文件中定义 * /
func (int x,y)
{… }
```

在F1.cpp和F2.cpp两个文件中都要使用a、b和c 3个变量。在F1.cpp文件中定义了a、b和c,在F2.cpp文件中用extern把3个变量声明为外部变量,表示这些变量已在其他文件中定义,并声明了这些变量的类型和变量名,编译系统不再为它们分配内存空间,而只利用声明的信息进行编译时的检查。

5.6.5 静态变量

静态变量通常是在变量定义时就分配存储单元,整个程序结束时被撤销。而自动存储类别的变量是在程序执行过程中,在进入它定义所在的程序块时才分配存储单元,退出语句块时立即释放。典型的例子是函数的形式参数,在函数定义时并不给形参分配存储单元,只是在函数被调用时,才予以分配,函数执行完毕立即释放。如果一个函数被多次调用,则反复地分配、释放形参变量的存储单元。总之,静态存储类别的变量是一直存在的,而动态存储类别的变量则时而存在时而撤销。

静态变量的类别描述符是static。静态变量采取静态存储方式,但是采取静态存储方式的量不一定就是静态变量。例如,外部变量虽然采取静态存储方式,但不一定是静态变量,必须由static加以限制后才能成为静态外部变量,或称静态全局变量。对于自动变量,前面

已经介绍它采取动态存储方式。但是,也可以用 static 限定它为静态变量,或称静态局部变量,也采取静态存储方式。例如:

```
static int a,b;
```

静态局部变量采取静态存储方式,具有以下特点。

(1) 静态局部变量在函数内定义,但不像自动变量那样,当执行该函数时就存在而退出函数时就撤销,静态局部变量始终存在着。其生存期为,从程序第一次执行到声明该变量的语句开始,到整个程序的执行完毕时结束。

(2) 静态局部变量的生存期虽然延续到整个程序执行完毕,但其作用域仍与自动变量相同,即只能在定义该变量的函数或语句块内使用该变量。退出该函数或语句块后,尽管该变量还继续存在,但其他的程序代码不能使用它,这是静态局部变量与全局变量的不同之处。

(3) 对于基本类型的静态局部变量,如果在声明时未赋予初值,则系统自动赋予 0 值,而对于自动变量,系统不会赋初值,其初始值是不定的。对于静态局部变量来说,虽然离开定义它的函数后不能使用,但如再次进入定义它的函数时,它又可继续使用,而且保存了前次被调用后留下的值。

软件工程提示:当多次调用一个函数,并要求在各次调用之间保留某些变量的值时,可考虑采用静态局部变量。虽然用全局变量也可以达到上述目的,但全局变量能够被所有的程序代码访问,有时会造成意外的副作用,而静态局部变量可以将对该变量的访问限制在函数或语句块内,因此仍以采用局部静态变量为宜。

考虑下面的程序:

```
int main()
{
    int i;
    void f();                    /* 函数声明 */
    for(i=1; i<=5; i++)
        f();                     /* 函数调用 */
    return 0;
}
void f()                         /* 函数定义 */
{
    auto int j=0;
    ++j;
    printf("%d\\n", j);
}
```

程序中定义了函数 f,其中的变量 j 说明为自动变量并赋予初始值为 0。当 main 中多次调用 f 时,j 均赋初值为 0,故每次输出值均为 1。下面把 j 改为静态局部变量,程序如下:

```
int main()
{
    int i;
```

```
        void f();                          /* 函数声明 */
        for(i=1; i<=5; i++)
            f();                           /* 函数调用 */
        return 0;
    }
    void f()                               /* 函数定义 */
    {
        static int j=0;
        ++j;
        printf("%d\\n", j);
    }
```

由于j为静态变量,能在每次调用后保留其值,并在下一次调用时继续使用,所以输出值成为累加的结果。

5.7 作 用 域

一个标识符能被使用的程序区域范围称为这个名字的作用域。生存期和作用域是从时间(程序执行的时间)和空间(源程序的范围)这两个不同的角度来描述变量的特性,二者既有联系又有区别。作用域是一个静态的概念,说明了一个名字在哪些代码中能够被使用,它开始于标识符的声明处,是程序中的一段区域,一个标识符只有在其作用域内才是可见的,才能被程序代码访问。而生存期是一个动态的概念,说明了名字表示的数据对象存在的时间区间。

C++作用域有6种:块作用域、函数作用域、函数原型作用域、文件作用域、类作用域和名字空间作用域。本节介绍前4种作用域,类作用域将在后面章节中讨论。

1) 块作用域(局部作用域)

在程序块(即用一对花括号括起来的语句,函数体也是一个块,一个控制结构语句也构成一个块)中声明的标识符具有块作用域,该标识符的作用域从声明点开始,到所在块结束处为止。一个在程序块B1中说明的名字X只在B1中有效(局部于B1);如果B2是嵌套在B1内部的一个内层块,且B2中对标识符X没有新的声明,则原来的名字X在B2中仍然有效;如果B2对X重新作了声明,则B2对X的任何引用都是指重新声明过的这个X。这就是所谓的"最近嵌套原则"。函数的形参具有块作用域,开始于函数定义开始的第一个左花括号处,结束于函数定义结束的右花括号处。

2) 函数作用域

具有函数作用域的标识符在该函数内的任何地方可见。在C++中只有标号具有函数作用域,它在整个函数范围内可见,包括标号定义之前和之后的任何代码。容易理解,标号在一个函数内必须唯一。

3) 函数原型作用域

函数原型作用域是C++程序中最小的作用域,是指在函数原型声明时形式参数的作用范围(开始于函数原型声明的左括号,结束于函数原型的右括号)。只有函数原型的形式参数表中使用的标识符才具有函数原型作用域。函数原型的参数表中的形式参数表只要求类

型,参数名可有可无,如果有参数名,该名字也只在该函数原型范围内有效。如果函数原型参数表中使用了参数名,那么编译器会忽略这些名字。

4) 文件作用域

在函数和类之外声明的标识符具有文件作用域,其作用域从声明点开始,在文件结束处为止。如果标识符出现在头文件的文件作用域中,那么当这个头文件被包括进其他源程序文件中时,该标识符的作用域将扩展到嵌入这个头文件的源程序文件中,直到该源程序文件结束。

由于自动变量的生存期都局限于定义它的函数体或复合语句(语句块)内,因此不同的语句块中允许使用同名的变量而不会混淆,即使在函数内定义的自动变量也可与该函数内部的复合语句中定义的自动变量同名。C++ 按照“最近嵌套原则”来确定每个标识符到底是哪个数据的名字。

【例 5-7】 变量的作用域和生存期示例。

```
//ex5_7.cpp: 变量的作用域和生存期示例
#include<iostream.h>

void a( void );          //函数原型
void b( void );          //函数原型
void c( int i );         //函数原型,i具有函数原型作用域,仅在该函数原型中可见

int x=1;                 //全局变量,具有文件作用域,从当前位置直到文件结束都可见

int main()
{
   int x=5;              //局部变量,具有块作用域,从当前位置直到 main结束可见

   cout<<"local x in outer scope of main is "<<x<<endl;

   {                     /* 开始新的块作用域 */
      int x=7;           //局部变量,具有块作用域,从当前位置直到当前复合语句结束可见

      cout<<"local x in inner scope of main is "<<x<<endl;
                         //访问的是内层声明的 x,外层声明的 x不可见
   }                     /* 新的块作用域结束 */

   cout<<"local x in outer scope of main is "<<x<<endl;

   a();                  //函数 a使用了局部自动变量 x
   b();                  //函数 b使用了局部静态变量 x
   c(10);                //函数 c使用了全局变量 x
   a();                  //函数 a中的局部自动变量 x被重新建立
   b();                  //局部静态变量 x保留了上次调用后的值
   c(10);                //全局变量 x保留了原来的值

   cout<<"local x in main is "<<x<<endl;
   return 0;
}

void a( void )
{
   int x=25;             //每次调用都重新建立并初始化
```

```
    cout<<endl<<"local x in a is "<<x<<" after entering a"<<endl;
    ++x;
    cout<<"local x in a is "<<x<<" before exiting a"<<endl;
    //结束时,撤销局部自动变量 x
}
void b( void )
{
    static int x=50;   //只在程序开时执行时初始化一次
    cout<<endl<<"local static x is "<<x<<" on entering b"<<endl;
    ++x;
    cout<<"local static x is "<<x<<" on exiting b"<<endl;
    //结束时,局部静态自动变量 x不被撤销
}
void c( int i )
{
    cout<<endl<<"global x is "<<x<<" on entering c"<<endl;
    x* =i;           //使用全局变量 x
    cout<<"global x is "<<x<<" on exiting c"<<endl;
}
```

程序执行结果:

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 50 on entering b
local static x is 51 on exiting b

global x is 1 on entering c
global x is 10 on exiting c

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 51 on entering b
local static x is 52 on exiting b

global x is 10 on entering c
global x is 100 on exiting c
local x in main is 5
```

5.8 递归函数

到目前为止,所介绍的函数调用都是按照层次方式,由一个函数调用另一个函数。C++是否允许函数调用自己呢? C++确实允许函数直接或间接地调用自己。这种直接调

用自己或者通过其他的函数间接地调用自己的函数称为递归函数(recursive function)。对某些问题,可以通过递归函数直观、简洁地加以解决。本节首先介绍递归的概念,然后通过具体的例子来学习设计递归函数的基本方法。

5.8.1 递归的概念

实际生活当中有许多这样的问题,这些问题比较复杂,问题的解决又依赖于类似问题的解决,只不过后者的复杂程度或规模较原来的问题更小,而且一旦将问题的复杂程度和规模化简到足够小时,问题的解法其实非常简单。许多数学问题表现得更突出。例如,计算某个自然数 n 的阶乘,有如下公式:

$$n!=n\times(n-1)\times(n-2)\times\cdots\times2\times1$$

对于这种阶乘的计算,可以利用下面循环结构:

```
int factorial=1;
for (int counter=1; counter<=n; counter++)
    factorial=factorial * counter;
```

可以从另外一个角度来看待阶乘的计算,自然数 n 的阶乘可以通过递归定义为:

$$\begin{cases} n!=n\times(n-1)! & n>1 \\ n!=1 & n=1 \end{cases}$$

根据这个递归定义,假设 $n=5$,那么 5! 的计算过程如图 5-2 所示。

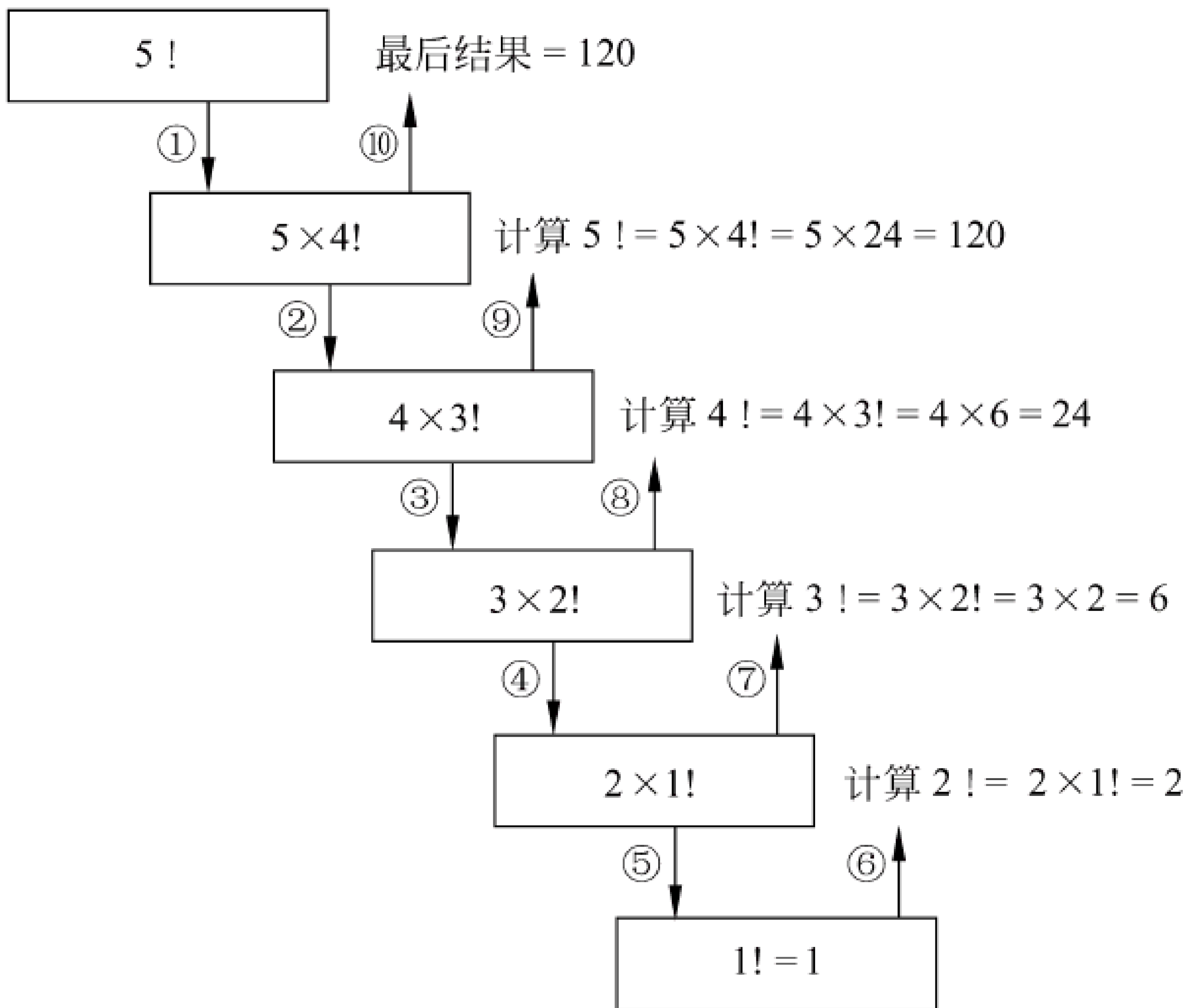


图 5-2　5!的递归计算过程

从图 5-2 中可以看出,为了计算 5!要先计算出 4!,要计算 4!又要先计算出 3!的结果,要得到 3!的结果则要先计算 2!,而要求 2!又需要先计算 1!。根据定义,1!为 1,有了 1!就可以计算 2!了,有了 2!的结果就可以计算出 3!的值,有了 3!的值就可以算出 4!的值,最后可以得到 5!的结果。这种解决问题的方法具有明显的递归特征。从这一递归计算过程可以看到,一个复杂的问题,被一个规模更小、更简单的类似的问题替代了,经过逐步分解,最

后得到了一个规模非常小、非常简单的、更容易解决的、类似的问题，将该问题解决后，再逐层解决上一级问题，最后解决了比较复杂的原始的问题。

很多问题具有上面这个例子的特征，都可以用递归方法加以解决。在程序设计时，这类问题求解的方法可以用递归函数来实现。在用递归函数解决问题时，递归函数可能只需解决最简单的情况，即所谓的“基本情况”。对于最基本的情况，递归函数往往只需要简单地返回一个结果。如果要解决更复杂的问题，那么函数要将问题分成两部分：一部分是当前能够直接处理的部分；另一部分是函数当前不能直接处理的部分，这部分子问题也可以转化成原问题求解，只是问题有所简化或缩小。由于这个子问题与原问题相似，因此该函数可以通过调用其自身来解决这个简化后的子问题，这称为“递归调用”(recursive call)。每一步递归的返回结果都要和函数前一部分(能够处理的部分)结合起来，得到当前问题的结果，并将结果返回给上一级。

实际上，递归函数的调用过程与一般函数的调用过程完全一样，正因如此，函数的递归调用有下面两个性质。

(1) 像一般函数调用一样，发生函数调用时，调用程序在函数调用处暂时挂起，程序控制离开调用程序转入被调用函数执行，只有当被调用函数执行完后，才返回到调用程序的调用处继续向下执行，所以调用函数一定要在被调用函数执行完成后才能继续执行并结束。在函数的递归调用过程中，表现出典型的“后进先出”的特征，最先进入的函数调用最后结束并返回，最后开始执行的函数调用最先结束并得到调用结果。

(2) 如同一般函数调用一样，每当一个函数被调用，系统就会为该函数的这次执行分配存储空间，包括为该函数的形式参数和局部变量分配单元。因此，递归函数执行时，在某一时刻，计算机内可能有该递归函数的多个活动同时存在，每个活动(即函数的每次调用执行)都有自己对应的存储空间，也就是说，函数的形式参数和局部变量，在函数的每次调用执行中都有不同的存储空间。

以上两点性质对于准确理解递归函数的执行过程具有重要的意义。

【例 5-8】 计算一个正整数 n 的阶乘。

```
//ex5_8.cpp:计算一个正整数 n的阶乘
#include<iostream.h>

long factorial(long);

int main()
{
    int i;

    cout<<"Please enter a positive integer: ";
    cin>>i;
    cout<<i<<"!="<<factorial(i)<<endl;

    return 0;
}

long factorial(long number)
{
    if (number<=1)
```

```
        return 1;
    else
        return ( number * factorial( number- 1 ) );
}
```

程序执行结果：

```
Please enter a positive integer: 10
10 != 3628800
```

注意：本例中 factorial 函数的形式参数 number，在 factorial 每次调用执行时都有不同的存储空间。

5.8.2　递归函数应用举例

本节通过两个例子深入学习递归函数的设计方法。

【例 5-9】　用递归方法求 n 阶勒让德多项式的值：

$$P_n(x)=\begin{cases}1 & n=0\\ x & n=1\\ ((2n-1)\cdot x\cdot P_{n-1}(x)-(n-1)\cdot P_{n-2}(x))/n & n>1\end{cases}$$

设计思想：采用递归程序设计方法时，与其说是设计问题的解法，不如说是将问题的递归性质分析清楚，用递归的方法描述问题。一般按照下面两步进行设计即可。

（1）准确分解问题的递归结构，分清楚哪些是当前能够直接处理的，哪些是当前不能直接处理的、而其解决方法又和原来的问题一样，只不过相对原问题简化一些。

（2）根据问题的递归结构特征规划递归算法。递归算法与递归问题是对应的：当前能直接解决的问题就设计相应的算法解决之；不能直接解决的，通过递归调用算法自身来解决。综合这两部分的结果就可得到问题的解决方法。

由于 n 阶勒让德多项式本身就是通过递归进行定义的，因此本例的递归结构是非常清楚的：当 n 等于 0 和 1 时，问题的答案可以直接计算得到；当 $n>1$ 时，当前问题的答案 $P_n(x)$需要 $P_{n-1}(x)$和 $P_{n-2}(x)$的结果才能计算。设计程序如下。

```
//ex5_9.cpp: 用递归方法求 n阶勒让德多项式的值
# include< iostream.h>

float P(int, float);

int main ()
{
    int n;
    float x;

    cout<< "Please enter n and x: ";
    cin>> n>> x;

    cout<< "P( "<< n<< ", "<< x<< " )= ";
    cout<< P(n, x)<< endl;

    return 0;
```

```
}
float P(int n, float x)
{
    if ( n==0 )
        return (1);
    else  if ( n==1 )
        return ( x );
    else
        return ( ( (2* n-1) * x* P( (n-1), x )-
                    (n-1) * P( ( n-2), x )) / n );
}
```

程序执行结果：

```
Please enter n and x: 3 7
P(3, 7)=847
```

这个例子中，问题本身就是一个递归函数，因此递归函数的设计非常直接而简单。有时候，很多问题的递归特点需要仔细分析才能发现，如下面的汉诺(Hanoi)塔问题。

【例 5-10】 假设有3根针A、B、C，其中A针上有N个盘子，盘子大小不等，大的在下，小的在上。要将N个盘子从A针移动到C针，每次只能移动一个，可借助B针，但必须保证任何时候3根针上盘子始终保持大盘在下、小盘在上。

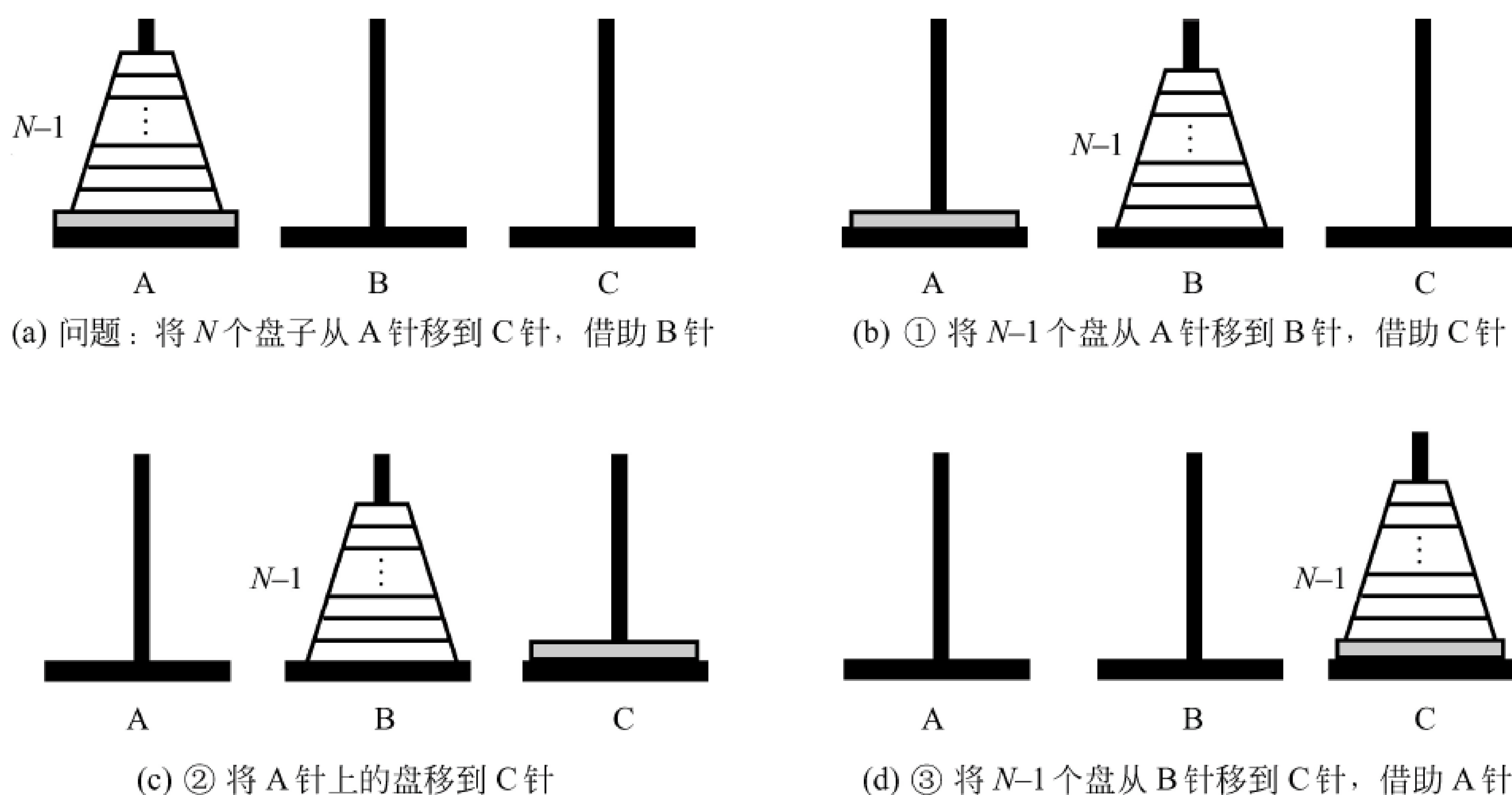
设计思想：汉诺塔是一个经典的递归问题。该问题如果采取非递归方法非常复杂，但是采取递归解法则很简洁。

如果盘子数目为1，那么搬动的方法很简单，即从A针移到C针即可；如果$N>1$，将N个盘子从A针移到C针，借助B针，可以分解成以下3个步骤。

① 将$N-1$个盘从A针移到B针，借助C针。

② 将A针上的盘移到C针。

③ 将$N-1$个盘从B针移到C针，借助A针，如图5-3所示。



(a) 问题：将N个盘子从A针移到C针，借助B针

(b) ① 将N–1个盘从A针移到B针，借助C针

(c) ② 将A针上的盘移到C针

(d) ③ 将N–1个盘从B针移到C针，借助A针

图5-3 汉诺塔移动示意图

注意：N 等于 1 时，可以直接解决；当 $N>1$ 时，算法的第①和③步与原问题相似，只是移动的盘数减少了。通过分析，发现了问题的递归性质。到目前为止，如果将完成上述算法的函数头设计如下(注意函数的参数与算法中数据的关系)。

```
void moveDisks(int diskNum, char sourcePole, char targetPole, char midPole)
```

各参数意义下：

diskNum　欲搬动的盘子的数目。

sourcePole　欲搬动的盘子原来所处的针的编号，即搬动的源针。

targetPole　盘子搬动的目标针的编号，即搬动的目标针。

midPole　盘子搬动的借助的针的编号，即搬动的借助针。

该函数的功能可描述为：将 diskNum 个盘子，从 sourcePole 针，借助 midPole 针，搬到 targetPole 针上。

将算法写成程序，就可以得到该函数的体。整个程序如下。

```
//ex5_10.cpp: 汉诺(Hanoi)塔问题的递归解法
#include<iostream.h>

void moveDisks(int diskNum, char sourcePole, char targetPole, char midPole);

int main()
{
    int n;

    cout<<"Please enter the number of the plates: ";
    cin>>n;

    if ( n<=0 )
        cout<<"The plates number must be greater than 0!";
    else
        moveDisks( n, 'A', 'C', 'B' );

   return 0;
}
/***************************************************************************
Function:     moveDisks

Parameters:
    diskNum      欲搬动的盘子的数目
    sourcePole   欲搬动的盘子原来所处的针的编号,即搬动的源针
    targetPole   盘子搬动的目标针的编号,即搬动的目标针
    midPole      盘子搬动的借助的针的编号,即搬动的借助针

Description:                 将 diskNum 个盘子,从 sourcePole 针,借助 midPole 针,搬到 targetPole 针上
***************************************************************************/
void moveDisks(int diskNum, char sourcePole, char targetPole, char midPole)
{
    if ( diskNum==1 )
```

```
            cout<<sourcePole<<"->"<<targetPole<<endl;
        else
        {
            //将 diskNum-1个盘从 sourcePole针移到 midPole针,借助 targetPole针
            moveDisks(diskNum-1, sourcePole, midPole, targetPole);
            //将 sourcePole针上的盘移到 targetPole针
            cout<<sourcePole<<"->"<<targetPole<<endl;
            //将 diskNum-1个盘从 midPole针移到 targetPole针,借助 sourcePole针
            moveDisks(diskNum-1, midPole, targetPole, sourcePole);
        }
}
```

程序执行结果:

```
Please enter the number of the plates: 3
A->C
A->B
C->B
A->C
B->A
B->C
A->C
```

如果输入的盘子数目太大,程序运行将花费相当长的时间。这一方面是因为问题复杂,另一方面也反映出递归函数调用由于在函数调用上的开销太大,因而效率较低。有的问题既可以用递归方法来解,也存在非递归解法,二者相比较各有优缺点:递归解法设计思想清晰简洁,程序易于理解和维护,但是执行效率可能较低;非递归解法设计思想复杂,不容易理解,不利于程序的维护,但执行效率较高。

5.9 预处理指令

第1章曾提到预处理功能,预处理发生在编译之前(参见图1-4),主要完成如下功能:把其他文件包含到当前文件中、定义符号常量和宏、程序代码的条件编译以及预处理指令的条件执行。所有的预处理指令都是用#开头的。一行预处理指令的前面只能出现空白字符。注意,预处理指令不是C++语句,因此也不能用分号结尾,预处理指令由预处理程序执行,在编译之前处理完毕,预处理的结果作为编译器的输入。预处理的结果不是目标代码,它只是对源程序进行简单处理后得到的高级语言程序,还需要经过编译器的翻译才能变成目标代码。

预处理指令主要包括:

#include、#define、#error、#if、#else、#elif、#endif、#ifdef、#ifndef、#undef、#line、#pragma。

每个预处理指令均带有符号#。下面介绍一些常用指令。

5.9.1 ＃include 指令

＃include 指令的作用是指示编译器将该指令所指出的另一个源文件嵌入 ＃include 指令所在的程序中，文件应使用双引号或尖括号括起来。例如：

```
#include<stdio.h>
```

表示将文件 stdio.h 包括到该预处理指令处。C++ 的程序也允许嵌入其他文件，例如：

```
main()
{
    #include<help.cpp>
}
```

假设 help.cpp 是另一个文件，其内容可为一行语句：

```
printf("Glad to meet you here!");
```

那么，上述函数经过预处理之后，变换成：

```
main()
{
    printf("Glad to meet you here!");
}
```

尽管＃include 指令可以包括各种源程序文件，但使用最多的是用来包括头文件。

在 C++ 语言中每一个标准库都有对应的头文件(header file)，该头文件包含了该库中各种数据类型、类、常量和函数原型的定义。

程序员可以建立自定义头文件，自定义头文件应以.h 结尾，并使用＃include 预处理指令将自定义的头文件包含到程序中。例如，在程序的开头用下列指令可把 const.h 头文件包含到程序中。

```
#include "const.h"
```

在＃include 指令中，文件名使用尖括号和双引号具有不同的意义。使用尖括号时，预处理程序总是在指定目录中寻找被包括的文件，通常标准库的头文件都用尖括号括起来，因为它们都在固定的目录中；使用双引号时，预处理程序将在被编译的程序所在的目录中寻找被包括的文件。

5.9.2 ＃define 指令

＃define 指令可以定义符号常量或宏。＃define 指令定义的一般形式是：

```
#define 符号常量标识符 字符串
#define 宏标识符 字符串
```

由＃define 指令定义后，在程序中每次遇到已定义的符号常量标识符或宏标识符时，就用所定义的字符串代替它。

例如,可用下面指令定义符号常量,其中TRUE表示数值1,FALSE表示0。

```
#define TRUE 1
#define FALSE 0
```

一旦在源程序中使用了TRUE和FALSE,编译时会自动地用1和0代替。

#define指令还可以定义宏。宏是在预处理指令#define中定义的一种操作。宏是C++从C中继承保留下来的。宏在C程序中使用很广,但由于C++提供了模板和内联函数,宏的作用减弱了。这里只简单介绍宏的定义和使用。和符号常量一样,程序中的宏标识符也在程序编译之前被替换文本取代。可以定义带参数或不带参数的宏。预处理器就像处理符号常量一样处理不带参数的宏。对带有参数的宏的处理方式是:先用替换文本取代参数,然后再把宏展开,即用替换文本取代程序中的标识符和参数表。

看一个定义宏的例子,假设有定义:

```
#define PI 3.1415926
#define CIRCLE_AREA(x)  (PI* (x)* (x))
```

如果程序中出现了下面的语句:

```
area=CIRCLE_AREA(5);
```

预处理程序将该语句展开为:

```
area= (3.1415926* (5)* (5));
```

用括号把宏的参数x括起来是为了保证在宏参数是表达式时,也能够强制编译器以正确的顺序计算表达式的值。例如,下列语句

```
area=CIRCLE_AREA(c+2);
```

展开成为:

```
area= (3.14159* (c+2)* (c+2));
```

表达式中的括号使得表达式能够以正确的顺序计算。如果去掉括号,宏就被展开为:

```
area=3.14159* c+2* c2;
```

就会错误地按照下面的表达式求值:

```
area= (3.14159* c)+ (2* c)+2;
```

还可以定义更复杂、功能更强大的宏,有兴趣的读者可以进一步深入思考并阅读相关资料。

程序设计技巧提示:宏和函数调用不同。宏的替换由预处理程序完成,发生在编译之前的预处理阶段;而函数调用发生在目标代码执行时。从执行效率上考虑,通过宏扩展产生的代码执行效率较高,相对于函数调用来说,省去了函数调用的开销,在这一点上与内联函数类似。从程序设计角度看,如果操作复杂,定义宏比定义函数难度更大,宏替换经常带来设计者意想不到的情况。

* 5.9.3 ＃if、＃else、＃endif、＃ifdef、＃ifndef 指令

＃if、＃else 和＃endif 指令为条件编译指令，它的一般形式为：

```
#if 常量表达式
    程序段 1;
#else
    程序段 2;
#endif
```

上述结构的含义是：若＃if 指令后的常数表达式为 true，则编译＃if 到＃else 之 间的程序段 1；否则，编译＃else 到＃endif 之间的程序段 2。

例如：

```
#define MAX 200
main()
{
    #if MAX> 999
          printf("compiled for bigger\n");
    #else
          printf("compiled for small\n");
    #endif
}
```

这段代码相当于：

```
#define MAX 200
main()
{
    printf("compiled for small\n");
}
```

如果将 MAX 定义为 2000，原来的代码相当于：

```
#define MAX 2000
main()
{
  printf("compiled for bigger\n");
}
```

＃ifdef、＃ifndef 指令分别相当于＃if define 和＃if !define，一般形式为：

```
#ifdef 宏替换名
#ifndef 宏替换名
```

其意义等价于：

```
#if define 宏替换名
```

```
#if !define 宏替换名
```

条件编译可以为程序设计带来很多好处。例如,下面的程序:

```
#ifdef DEBUG
printf("Show debug message here!\n");
#endif
```

只有当定义了符号DEBUG时,才会编译该输出语句;否则,目标程序中不会有该语句对应的代码。这样,在调试程序时可以加入许多输出调试信息的语句,当程序调试通过后,在程序中删除符号DEBUG的定义指令或使用#undef DEBUG,再进行编译,这时得到的目标代码就不会包含输出调试信息的代码了。

程序设计技巧提示:使用条件编译,可以用通过定义的符号来选择程序的功能。例如,有时候可以写一个功能丰富的程序,但是根据应用需求的不同,可能需要选择定制程序的功能,这时可以通过定义符号来选择编译哪一部分代码,而不必在编写程序时,大量删除某些代码,或添加或取消大量的注解。

习　题　5

5.1　回答以下问题:

(1) 函数的定义格式是如何规定的?在具体定义时,哪些是可以省略的?

(2) 在什么情况下,必须使用函数的原型声明?

(3) 如何实现函数的返回值?用返回值实现函数间的数据传递有何局限性?

(4) 什么叫内联函数?如何定义内联函数?使用内联函数的实质与目的是什么?

(5) 什么是函数的重载?调用重载函数时,通过什么来区分不同的重载函数?

(6) 什么叫递归?递归和递推有什么区别和联系?

(7) 编译预处理指令是C++语句吗?为什么?

(8) 变量的存储类别有哪几种?其作用域和生命期各有什么不同?

(9) 静态变量有何特点?

(10) 什么是传引用调用?它有何特点?

5.2　从键盘上输入10个浮点数,求出它们的和以及平均值,要求用函数实现。

5.3　设计一个函数,这个函数有两个形参,一个表示年份,一个表示月份,要求该函数返回这月的天数。

5.4　编写一个函数求两个整数最小公倍数。

5.5　求400之内的亲密对数。所谓亲密对数,即A的所有因子之和等于B,B的所有因子之和等于A。

5.6　设计一个函数,实现把输入的一个十进制数转换为十二进制数。

5.7　编写函数multiple,确定一对整数中的第二个整数是否为第一个整数的倍数。函数取两个整数参数,如果第二个整数是第一个的倍数则返回true,否则返回false。在程序中输入一系列整数,并调该函数。

5.8　设计一个函数primc,这个函数带有一个整型参数,当这个参数的值是素数时,该函数返回非0,否则返回0。利用这个函数编写一个程序来验证哥德巴赫猜想:任何一个充分大的偶数(大于等于6)总可以表示成两个素数之和。

5.9　设计一个程序,用重载函数实现计算两个数的平方和,分别实现整型和浮点型的计算。

5.10　编写一个C++程序,用内联函数circleArea提示用户输入圆的半径,并计算和打印圆的面积。

5.11　对于如下程序,指出下列元素的作用域(函数范围、文件范围、块范围或函数原型范围)。

(1) main 中的变量 x。
(2) cube 中的参数 y。
(3) 函数 cube。
(4) 函数 main。
(5) cube 的函数原型。
(6) cube 函数原型中的标识符 y。

```
#include<iostream.h>
int cube(int y);
int main(int y);
{
    int x;
    for (x=1;x<=10;x++)
        cout<<cube(x)<<endl;
return 0;
}
int cube(int y)
{
    return y*y*y;
}
```

5.12 已知组合数：$c(m,r)=m!/(r!(m-r)!)$，其中 m、r 为正整数，且 $m>r$。分别求出 $c(5,2)$、$c(8,6)$ 的组合数。阶乘及组合数用函数完成。

5.13 用牛顿迭代法设计一个通用的解一元二次方程的函数。求方程 $x^3-3x^2+1=0$，$3x^2-4x+2=0$ 的根。牛顿迭代公式：$x_{n+1}=x_n-f(x_n)/f'(x_n)$，其中，$f'(x)$ 是 $f(x)$ 的导函数。迭代初值由键盘输入。

5.14 编写一个递归函数，将一个正整数的每一位取出来并按从低位到高位的顺序输出。

5.15 根据斐波那契数列的计算公式，用递归的方法计算该数列。

5.16 将例 5-2 中用辗转相除方法求最大公因子的函数改写成递归函数，试比较这两种不同实现方法。

5.17 当 $x>1$ 时 Hermite 多项式定义为：

$$H_0(x)=1$$
$$H_1(x)=2x$$
$$H_n(x)=2xH_{n-1}(x)-2(n-1)H_{n-2}(x)$$

试编写输出 Hermite 多项式对应变量 x 的前 n 项值的递归函数和非递归函数。

5.18 编写一个函数 power(float x,int n)用递归的方法计算 x 的 n 次幂。在主函数中实现输入输出。

5.19 已知三角形的三条边 a、b、c，分别用带参数的宏和函数编写求三角形的面积的程序。公式为：

$$s=(a+b+c)/2$$
$$\text{area}=\sqrt{s(s-a)(s-b)(s-c)}$$

5.20 写出下列程序的运算结果。

```
#include<iostream.h>
#define A b*b-b
#define C A-A
int main()
{
   int b=1;
```

```
    cout<< "C= "<< C;
    return 0;
}
```

5.21 使用#define 语句对求两个整数的最大值进行宏定义,并编写完整的程序:给定两个整数,求两个整数的最大值。

5.22 编写一个递归函数求斐波那契数列的前 n 项和,并在程序中加入条件编译语句,使得在定义了 DEBUG 时,显示调试信息,没有定义时则不显示。

第 6 章 数 组

【学习内容】

本章介绍 C++ 的数组。主要内容包括：

◆ 一维数组的定义与使用。

◆ 多维数组的定义与使用。

◆ 字符串与字符数组。

【学习目标】

◆ 理解数组的概念，掌握数组声明和使用的方法。

◆ 理解数组元素的存放方式。

◆ 熟练掌握数组和循环配合的程序设计方法。

程序设计过程中，除了要处理整型、实型、字符型、布尔型等简单数据类型的数据外，还需要处理具有结构性质的数据。例如，全班同学的学号、所有同学某次考试的成绩等，这些数据逻辑意义相关，有明显的结构特征。虽然这些数据本身都能够通过简单数据类型来表示，但是仅使用简单数据类型，不能很好地体现这些数据的逻辑相关性和结构特征。因此，需要一种机制来描述这些相关的数据，这就是数据结构。所谓数据结构，是指由简单类型的数据构造复合类型数据的方法和表示。把简单的数据类型的数据(如整数、实数和字符)加以组合，构造复合类型的数据(如数组、结构、联合等)。把复合类型的数据再加以组合，可构造更为复杂的复合类型的数据。

1966 年，N. 沃思和 C. A. R. 霍尔提出了数据结构的概念。随后，沃思在 Pascal 程序设计语言中提出数据类型的构造方法。1972 年，霍尔进一步阐述数据结构，并对每种结构进行非形式的描述，讨论了建立数据结构的几种方法。数据结构概念提出之后，高级程序设计语言大多提供了基本数据类型，如整型、逻辑型(布尔型)、实型和字符型，也提供如何由基本数据类型构造复合数据类型的方法。如 PL/1 中的“结构型”，Pascal 中的“记录型”等。C++ 作为高级程序设计语言，也提供了这种构造能力。

本章将介绍数组这一基本的数据结构。数组是由类型相同、逻辑意义相关的一组数据构成的。例如，上面提到的全班同学的学号，全班所有同学某次考试的成绩等，这些数据都可以用数组来描述。每个数组都有一个由标识符来充当的名字，也称为数组名。数组中的每个元素称为数组元素，它们按顺序分配在内存中一片连续的内存单元中，每个元素占用相同数目的单元。数组元素是按顺序排列的，顺序号称为数组元素的下标。数组元素通过数

组名和下标来表示。数组可以有多个下标,下标的个数称为数组的维数。数组按维数多少可分为一维数组、多维数组,按数组元素的类型可分为字符数组、整型数组、实型数组和指针数组等。

6.1 一维数组

6.1.1 一维数组的声明与初始化

1. 声明

一维数组只有一个下标。一般形式为:

```
<存储类别><元素类型><数组名>[<元素个数>];
```

＜存储类别＞与变量声明中的存储类别相同;＜元素类型＞指明了数组元素的类型,可以是整型、实型、布尔型和字符型等简单数据类型,也可以是用户定义的复合数据类型,包括数组类型;＜数组名＞由标识符充当,代表了一个数组变量;＜数组名＞后面的方括号是必须要有的,它说明前面的标识符是数组名,在C++中[]是作为运算符处理的,其优先级与括号相同,从左向右结合;方括号中的＜元素个数＞是整常量表达式,可以省略,表示数组元素的个数,注意数组元素的下标是从零开始计数的。例如:

```
int a[10];
```

声明了a是一个整型数组,它有10个整型元素,下标的变化范围是0～9。

数组元素是通过数组名和下标来访问的。程序中数组元素的表示形式为:

```
<数组名>[<下标表达式>]
```

＜下标表达式＞是值为整型的表达式,它指明了拟访问的数组元素的下标。例如,数组a的10个元素可以依次表示为:

```
a[0],a[1],a[2],…, a[9]
```

如果访问数组元素时提供的下标超出了元素的数目,会导致访问越界错误。C++不检查访问越界,如果发生了访问越界,有可能导致程序的运行时错误,这一点要特别注意。

数组a的各元素按顺序分配在内存中一片连续的内存单元中,如图6-1所示。

a[0]	−321
a[1]	2223
a[2]	87
a[3]	−231
a[4]	0
a[5]	92
a[6]	−171
a[7]	−612
a[8]	210
a[9]	32

图6-1 数组a[10]的内存分配

数组作为函数的形式参数或引用说明时,通常在声明时不指明元素的数目,而仅仅说明该参数是一个数组,其元素的数目由实在参数决定。

一维数组类似于数学中的向量。

2. 初始化

在声明数组时可以对数组中开始的若干个元素乃至全部元素进行初始化,其方法是把初始值按顺序放在花括号中,数

值之间用逗号分开。例如：

```
int a[10]={4};
float r[20]={0.1, 5.1};
double d[3]={10.0,5.0,1.0};
```

如果初始值的数目小于数组元素的数目，数组剩余的元素被自动初始化为0。例如：

```
int n[5]={0};
```

实际上将所有的数组元素都初始化为0。

如果初始值的数目超过了数组中元素的数目，编译程序将报告语法错误。

如果声明数组时省略了数组元素的数目，那么系统将根据初始值的数目来确定数组元素的数目，这时数组元素的个数与初值的个数相同。例如：

```
int x[]={ 1, 2, 3, 4, 5 };
```

实际上声明了一个包含了5个整型元素的数组x，并对每个数组元素使用初始值进行了初始化，各元素的值为：x[0]=1，x[1]=2，x[2]=3，x[3]=4，x[4]=5。

6.1.2　数组的应用

在使用上，数组元素和一般变量一样，可以构成表达式，参加各种运算。例如：

```
a[7]=4;              //给数组元素 a[7]赋值
x=2* a[i+2];         //数组元素作表达式的运算量
```

在实际编程过程中，通常把数组和循环语句结合起来，对数组元素进行遍历和处理。

【例6-1】　输出数组的每个元素，并对数组元素求和。

```
//ex6_1: 输出数组的每个元素,并对数组元素求和
#include<iostream.h>

#define SIZE 10

int main()
{
    int n[ SIZE ]={ 12, 34, 55, 71, 1, 65, 423, 19, 540, 10 };
    int i, sum=0;

    cout<<"Element"<<"\t Value";

    for ( i=0; i<=SIZE-1; i++) {
      cout<<"n["<<i<<"]="<<n[ i ]<<endl;
      sum+=n[ i ];
    }

    cout<<"The summary is: "<<sum<<endl;
    return 0;
}
```

程序执行结果：

```
Element Value
n[0]=12
n[1]=34
n[2]=55
n[3]=71
n[4]=1
n[5]=65
n[6]=423
n[7]=19
n[8]=540
n[9]=10
The summary is: 1230
```

下面分析一个稍复杂的问题：数组元素排序。在很多应用中，都需要将数组里面的元素按照某个标准排好顺序。这是一个经典的问题，已经有了许多排序算法，如直接选择排序、冒泡排序等。这里先学习直接选择排序方法，如果希望了解更多排序算法，可以参看数据结构和算法方面的教材。

直接选择排序是一种简单的排序方法。例如，要将一个包含 n 个元素的数组的各元素从小到大排序，其工作过程是：首先在未排序序列中找到最小元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小元素，放到排序序列末尾。以此类推，直到所有元素均排序完毕。此时，数组中所有元素就已经按照从小到大的顺序排列好了。图 6-2 是一个直接选择排序的示意图。

```
排序前：       18   35    36    61    9     112   77    12
第 1 次扫描后：9    [35   36    61    18    112   77    12]
第 2 次扫描后：9    12    [36   61    18    112   77    35]
第 3 次扫描后：9    12    18    [61   36    112   77    35]
第 4 次扫描后：9    12    18    35    [36   112   77    61]
第 5 次扫描后：9    12    18    35    36    [112  77    61]
第 6 次扫描后：9    12    18    35    36    61    [77   112]
第 7 次扫描后：9    12    18    35    36    61    77    [112]
排序后：       9    12    18    35    36    61    77    112
```

图 6-2　直接选择排序过程示例

图中方括号括住的部分为尚未排序的部分，方括号之前为已处理的部分。如果数组有 n 个元素，在进行第 i 次扫描时，需通过 $n-i$ 次比较选出所需的元素(即第 i 小的元素)。直接选择排序总是需要对数组进行 $n-1$ 次扫描，就可以将所有的元素排好序。

【例 6-2】 使用直接选择排序方法对整数数组元素按照从小到大顺序排序。

```
//ex6_2:使用直接选择排序方法对整数数组元素按照从小到大顺序排序
#include< iostream.h>

#define SIZE 8

int main()
{
```

```
    int a[ SIZE ]={ 18,35,36,61,9,112,77,12};

    for (int i=0; i< SIZE - 1; i++)
    {
        //在未排序序列中找到最小元素
        int min=i;
        for (int j=i+1; j<SIZE; j++)
            if(a[j]<a[min])
                min=j;
        //将找到的最小元素与第 i个元素交换,即存放到排序序列末尾
        int temp=a[i];
        a[i]=a[min];
        a[min]=temp;
        //输出当前结果
        cout<<"After No. "<<i+1<<" scan:";
        for (int index=0; index<=SIZE-1; index++)
            if ( index==i+1 )
                cout<<"\t"<<"["<<a[ index ];
            else
                cout<<"\t"<<a[ index ];
        cout<<"]"<<endl;
    }
    //输出排序后的结果
    cout<<"After sorting: \t";
    for ( int index=0; index<=SIZE-1; index++)
        cout<<"\t"<<a[ index ];

    cout<<endl;

    return 0;
}
```

程序执行结果:

```
After No.1 scan: 9   [35    36    61    18    112    77    12]
After No.2 scan: 9   12     [36   61    18    112    77    35]
After No.3 scan: 9   12     18    [61   36    112    77    35]
After No.4 scan: 9   12     18    35    [36   112    77    61]
After No.5 scan: 9   12     18    35    36    [112   77    61]
After No.6 scan: 9   12     18    35    36    61     [77   112]
After No.7 scan: 9   12     18    35    36    61     77    [112]
After sorting:   9   12     18    35    36    61     77    112
```

程序中有二重循环,外循环控制对数组进行扫描的次数,内循环实现在“第 i 次扫描中,通过 $n-i$ 次比较选出所需的元素(第 i 小的元素)”。总体上,对数组的扫描次数为 SIZE－1。

6.1.3 数组作为函数参数

数组可以作为函数的参数进行传递。先看一个程序。

【例 6-3】 假设一个整型数组中存放了一个班10名同学的百分制成绩,由于评分制的调整,需要将百分制的成绩都转换为5分制。请编写程序实现这一功能,要求将分数改变作为函数实现,将要修改的数组作为函数的参数进行传递。

```
//ex6_3: 数组作为函数的参数
#include<iostream.h>
#define SIZE 10
void convertScores(int [], int);
int main()
{
    int scores[SIZE]={85, 63, 72, 52, 95, 82, 77, 69, 88, 73};
    //调用函数 convertScores
    convertScores(scores, SIZE);
    //输出调用函数 convertScores 后 scores 的情况
    cout<<"After calling convertScores: "<<endl;
    for (int index=0; index<=SIZE-1; index++)
      cout<<"scores["<<index<<"]="<<scores[index]<<endl;
    return 0;
}
void convertScores(int s[], int len)
{
   for ( int index=0; index<=len-1; index++)
     s[ index ]=s[ index ] / 20;
}
```

程序执行结果:

```
After calling convertScores:
scores[0]=4
scores[1]=3
scores[2]=3
scores[3]=2
scores[4]=4
scores[5]=4
scores[6]=3
scores[7]=3
scores[8]=4
scores[9]=3
```

程序中,convertScores是一个使用数组作为参数的函数,有两个参数:s是一个数组类型的形式参数,len是整型参数。要让函数通过函数调用接收数组,函数的参数表应指定接

收数组(如 s)。数组形参的方括号中的数组长度不是必需的,如果有,编译器也会将其忽略。为了防止访问越界,通常还要将数组元素的数目作为另一个参数一起传递过来。函数体内对数组 s 进行了遍历并将所有的元素都进行了转换。

主程序将数组 scores 作为参数传递给函数 convertScores,提供了相应的实在参数:不带方括号的数组名 scores 和数组的长度 SIZE。这个调用导致了函数 convertScores 被执行,最后输出的结果显示数组 scores 的每个元素都被转换了。这个例子说明,在被调用函数体内对作为形式参数的数组 s 的元素进行的修改操作发生在实在参数数组 scores 的元素上,这是否与前面介绍的 C++ 传递参数采取的传值方式相矛盾呢?是不是数组传递时采取的是传引用呢?实际上,C++ 处理数组参数时严格地采取了传值方式。

传递数组的方式有特殊的地方,要想准确理解传递数组的机制,首先要明确下面两点。

(1) 和一般变量一样,数组名也是有值的,数组名的值就是数组的第一个元素的地址。

(2) 和一般变量一样,数组作为函数参数进行传递时,传递的也是数组变量的值,即数组第一个元素的地址。

从上面的分析可以看到,正是由于数组名的值就是数组的第一个元素的地址以及访问数组元素的方式,才使得将数组传递给函数时尽管采取的是传值方式,但客观上却起到了传引用的效果。所以,上面的程序中实参 scores 的每个元素都发生了修改。

如果将数组元素作为参数传递给函数,与一般变量没有任何区别,都是传值。

下面的示例程序将例 6-2 的排序算法设计成一个独立的函数,同时提供一个输出数组元素的函数。这两个函数针对不同的数组类型进行重载,可提供给其他程序使用。

【例 6-4】 用函数实现数组元素排序和输出。

```
//ex6_4: 用函数实现数组元素排序和输出
#include<iostream.h>

#define SIZE 8

void sortArray(int [], int);
void sortArray(double [], int);
void displayArray(int [], int);
void displayArray(double [], int);

int main()
{
    int ai[SIZE]={18, 35, 36, 61, 9, 112, 77, 12};
    double af[SIZE]={12.1,-23.8, 3.7,-16.0, 9.1, 12.12, 7.7, 56.3};

    //调用函数 displayArray 输出 ai 排序前的数据
    cout<<"Before sorting:\n";
    cout<<"ai: \t";
    displayArray(ai, SIZE);

    //调用函数 sortArray 对 ai 进行排序
    sortArray(ai, SIZE);

    //调用函数 displayArray 输出 ai 排序后的结果
    cout<<"After sorting:\n";
```

```
        cout<<"ai: \t";
        displayArray(ai, SIZE);
        //调用函数 displayArray 输出 af 排序前的数据
        cout<<"Before sorting:\n";
        cout<<"af: \t";
        displayArray(af, SIZE);
        //调用函数 sortArray 对 af 进行排序
        sortArray(af, SIZE);
        //调用函数 displayArray 输出排序结果
        cout<<"After sorting:\n";
        cout<<"af: \t";
        displayArray(af, SIZE);
        return 0;
    }
    void sortArray(int b[], int len)
    {
        for (int i=0; i<len-1; i++)
        {
            //在未排序序列中找到最小元素
            int min=i;
            for(int j=i+1; j<len; j++)
                if(b[j]<b[min])
                    min=j;
            //将找到的最小元素与第 i 个元素交换,即存放到排序序列末尾
            int temp=b[i];
            b[i]=b[min];
            b[min]=temp;
        }
    }
    void sortArray(double b[], int len)
    {
        for (int i=0; i<len-1; i++)
        {
            //在未排序序列中找到最小元素
            int min=i;
            for(int j=i+1; j<len; j++)
                if(b[j]<b[min])
                    min=j;
            //将找到的最小元素与第 i 个元素交换,即存放到排序序列末尾
            double temp=b[i];
            b[i]=b[min];
            b[min]=temp;
        }
    }
    void displayArray( int b[], int len)
    {
        for ( int index=0; index<=len-1; index++)
```

```
            if ( index != len-1 )
                cout<<b[ index ]<<"\t";
            else
                cout<<b[ index ]<<endl;
}

void displayArray( double b[], int len)
{
        for ( int index=0; index<=len-1; index++ )
            if ( index != len-1 )
                cout<<b[ index ]<<"\t";
            else
                cout<<b[ index ]<<endl;
}
```

程序执行结果：

```
Before sorting:
ai:     18      35      36      61      9       112     77      12
After sorting:
ai:     9       12      18      35      36      61      77      112
Before sorting:
af:     12.1    -23.8   3.7     -16     9.1     12.12   7.7     56.3
After sorting:
af:     -23.8   -16     3.7     7.7     9.1     12.1    12.12   56.3
```

下面介绍一种在数组中查找某个元素的方法。

【例 6-5】 用二分法实现数组元素的查找。

设计思想：在数组中查找一个指定的元素有很多方法。最简单的一种是所谓的线性查找，即从头至尾对数组的每个元素进行扫描，逐个进行比较，直到找到指定的元素。下面介绍一种更高效的查找算法——二分法。二分法采取的是一种递归的思想。

（1）首先要求数组元素已经按从小到大顺序进行排列。

（2）把数组中央的元素（middle）和关键值（key）进行比较：

- 如果相等，则找到该元素。
- 如果 key＜middle，则在数组的前一半的元素中进行二分查找。
- 如果 key＞middle，则在数组的后一半的元素中进行二分查找。

显然，可以用一个递归函数实现上面的二分查找算法。为了能够形象地显示二分查找的进程，在程序的输出中给出了每次比较查找的范围和被比较的元素。

```
//ex6_5.cpp  用二分法实现数组元素的查找
#include<iostream.h>
#define SIZE 10

int binarySearch( int [],  int, int, int );
void displayArray( int [], int, int, int );

int main()
```

```
{
   int a[ SIZE ], i, key, result;

   for ( i=0; i<=SIZE-1; i++ )
      a[ i ]=2* i;

   cout<<"Please enter a number: ";
   cin>>key;

   result=binarySearch( a, key, 0, SIZE-1 );

   if ( result!=-1 )
      cout<<key<<" found in the array at index "<<result<<"."<<endl;
   else
      cout<<key<<" not found in the array."<<endl;

   return 0;
}
//二分查找
int binarySearch( int b[], int searchKey,
                        int low, int high )
{
   int middle;

   while ( low<=high ) {
      middle=( low+high ) / 2;

      displayArray( b, low, middle, high );

      if ( searchKey==b[ middle ] )
         return middle;
      else if ( searchKey<b[ middle ] )
         high=middle-1;
      else
         low=middle+1;
   }

   return -1;                              /* searchKey not found* /
}
//显示数组
void displayArray( int b[], int low, int mid, int high)
{
     for ( int i=0; i<=SIZE-1; i++ )
     {

         if ( i==low)
               cout<<"[";                  //标记搜索的起点
         if ( i==mid )
               cout<<"* ";                 //标记被比较的元素
         cout<<b[ i ];
         if ( i==high)
               cout<<"]";                  //标记搜索的终点
         if ( i !=SIZE-1)
```

```
            cout<< "\t";
        }
        cout<< endl;
}
```

程序执行结果：

```
Please enter a number: 7
[0    2    4    6    * 8    10    12    14    16    18]
[0    * 2    4    6]    8    10    12    14    16    18
0    2    [ * 4    6]    8    10    12    14    16    18
0    2    4    [ * 6]    8    10    12    14    16    18
7 not found in the array.

Please enter a number: 10
[0    2    4    6    * 8    10    12    14    16    18]
0    2    4    6    8    [10    12    * 14    16    18]
0    2    4    6    8    [ * 10    12]    14    16    18
10 found in the array at index 5.
```

对于一个有 N 个元素的数组，二分查找最多需要进行 $\log_2 N+1$ 次比较，而线性查找则最多需要 N 次比较。读者自己可写一个线性查找的函数，并和二分查找的函数进行比较。

6.2　多维数组

前面学习的一维数组可以看成是数学上的向量在计算机中的表示方式，那么数学上的矩阵如何表示呢？在 C++ 中，也可以构造多维数组来表示矩阵。

6.2.1　多维数组的定义与初始化

1. 多维数组的定义

C++ 中有多个下标的数组称为多维数组。具有两个下标表示的数组称为二维数组。多维数组可以有多于两个的下标，C++ 编译器支持至多 12 个下标的数组。例如：

```
int a[3][4];
char c[4][3][5];
```

声明 a 是一个整型二维数组，c 是字符型三维数组。二维数组是多维数组中最简单、最常用的多维数组。数学上的二维矩阵可以用二维数组表示出来。下面仅讨论二维数组，多维数组也可以类似地使用。

在数学上，二维矩阵可以看成是元素是向量的向量，在 C++ 中，二维数组也可以看成元素是一维数组的一维数组。上面声明的二维数组 a 有两个下标。第一个下标称为行，变化范围是 0～2；第二个下标称为列，变化范围是 0～3。因此，a 共有 3 行 4 列，12 个元素，其逻辑结构参见图 6-3。每个数组元素用数组名和两个下标表示。例如：

```
a[1][2],a[2][1]
```

分别表示第1行第2列的元素和第2行第1列的元素。

多维数组在内存中“按行”存放,越靠后的下标先变化,越靠前面的下标后变化。上面声明的二维数组a在内存中的存放顺序参见图6-4。

第0行	a[0][0]	a[0][1]	a[0][2]	a[0][3]
第1行	a[1][0]	a[1][1]	a[1][2]	a[1][3]
第2行	a[2][0]	a[2][1]	a[2][2]	a[2][3]
	第0列	第1列	第2列	第3列

图6-3　二维数组逻辑结构示意图

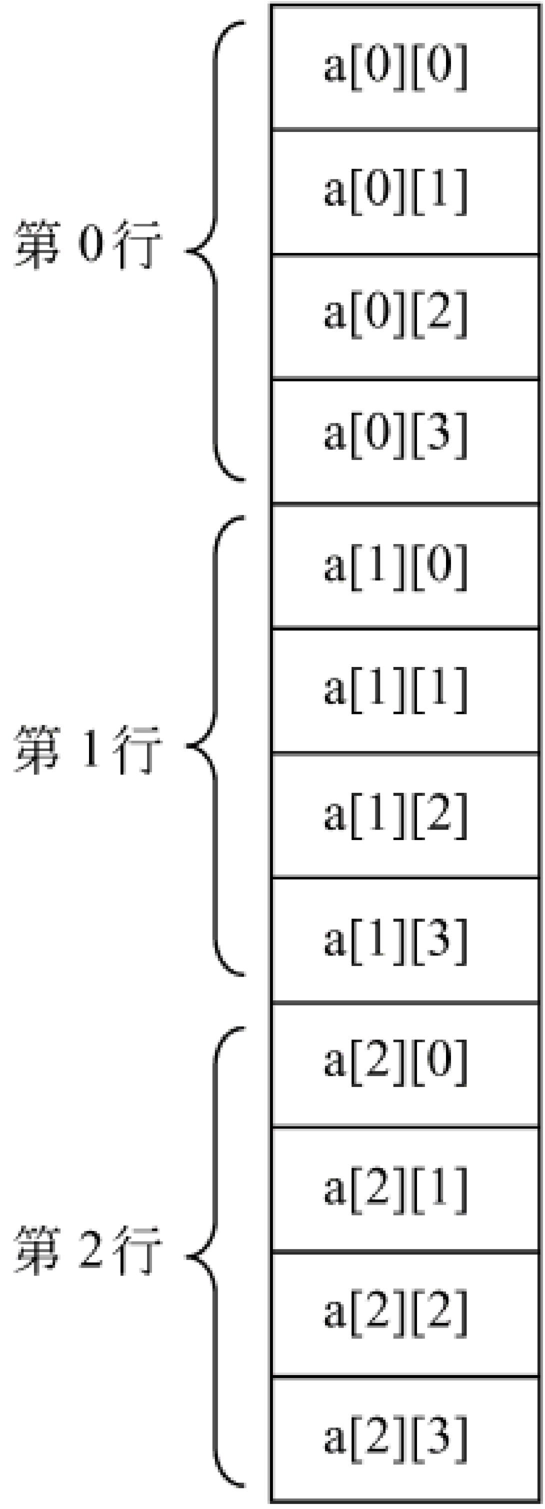


图6-4　二维数组存放示意图

2. 多维数组的初始化

多维数组也可以在声明时初始化,与一维数组的初始化一样,二维数组初始化时可以给全部元素赋初值,也可以只给部分元素赋初值。例如,一个2×2的二维数组matrix可以说明和初始化如下:

```
int matrix[2][2]={ {1, 2}, {3, 4}};
```

数组元素的值用花括号按行分组,因此整常数1和2初始化了matrix[0][0]和matrix[0][1],整常数3和4初始化了matrix[1][0]和matrix[1][1]。如果指定行没有足够的初始值,与一维数组类似,则该行的剩余元素初始化为0,如果初始值只给出了部分行,则剩余的行中的所有元素都被初始化为0。这样,下列说明:

```
int matrix[2][2]={ {1}, {3, 4}};
```

初始化完成后:matrix[0][0]为1,matrix[0][1]为0,matrix[1][0]为3,matrix[1][1]为4。而

```
int matrix[3][4]={ {1}, {3, 4}}
```

初始化完成后,数组matrix的各元素的值为:matrix[0][0]为1,matrix[1][0]为3,matrix[1][1]为4,其余元素均为0。

如果初始值之间没有用花括号按行分组,那么编译器会自动地用初始值顺序初始化第0行的元素、第1行的元素……显然,这种初始化的顺序是和数组元素的存放顺序相一致

的。如果初始值的数目少于数组元素的数目,剩余的元素自动初始化为0。例如:

```
int matrix[3][4]={ 1, 2, 3, 4, 5, 6 }
```

初始化完成后,数组a的各元素的值为:matrix[0][0]为1,matrix[0][1]为2,matrix[0][2]为3,matrix[0][3]为4,matrix[1][0]为5,matrix[1][1]为6,其余元素均为0。

6.2.2 应用示例

下面用二维数组来实现数学上的方阵(即行和列数目相同的矩阵)的加法、减法和乘法。矩阵上的各种运算都由相应的函数来实现,矩阵的输出也由专门的函数来完成。

与一维数组一样,多维数组变量的值也是数组第一个元素的地址。如果要将二维数组作为参数传递给函数,定义相应的形式参数时,要给出第二维的大小。

【例6-6】 用二维数组实现二维方阵的加法和乘法。

设计思想:采用逐步求精方法进行设计。首先,顶层的算法是:

1. 二维方阵初始化。
2. 两个二维矩阵进行加法和乘法运算,并输出结果:
   2.1　输出参加运算的两个方阵。
   2.2　执行方阵的运算。
   2.3　输出运算的结果方阵。

为了提高程序的模块化程度,可以将输出方阵的功能用单独的函数displayMatrix实现,方阵的加法和乘法运算也分别用addMatrix和mulMatrix两个函数实现。这样,在addMatrix和mulMatrix函数中只要按照矩阵加法和乘法定义,借助多重循环结构就可以实现相应的运算。

```
//ex6_6.cpp  用二维数组实现二维方阵的加法和乘法
#include<iostream.h>
#define SIZE 4
void addMatrix( int [][SIZE], int [][SIZE], int [][SIZE]);
void mulMatrix( int [][SIZE], int [][SIZE], int [][SIZE]);
void diaplayMatrix( int [][SIZE]);
int main()
{
    int m1[ SIZE ][ SIZE ]={  {1, 1, 1, 1},
                              {2, 2, 2, 2},
                              {3, 3, 3, 3},
                              {4, 4, 4, 4} };
    int m2[ SIZE ][ SIZE ]={  {5, 5, 5, 5},
                              {6, 6, 6, 6},
                              {7, 7, 7, 7},
                              {8, 8, 8, 8}};
    int resultAdd[ SIZE ][ SIZE ] ={ 0 };          //加法的结果
    int resultMul[ SIZE ][ SIZE ] ={ 0 };          //乘法的结果
```

```
    cout<<"***************************************************************";
    cout<<endl;
    //调用函数 diaplayMatrix 输出 m1 和 m2
    diaplayMatrix(m1);
    cout<<"                    +"<<endl;
    diaplayMatrix(m2);
    //调用函数 addMatrix 计算 m1+m2
    addMatrix(m1, m2, resultAdd);
    //调用函数 diaplayMatrix 输出 m1+m2 的结果 resultAdd
    cout<<"                    ="<<endl;
    diaplayMatrix(resultAdd);
    cout<<endl;
    cout<<"***************************************************************";
    cout<<endl<<endl;
    //调用函数 diaplayMatrix 输出 m1 和 m2
    diaplayMatrix(m1);
    cout<<"                     * "<<endl;
    diaplayMatrix(m2);
    //调用函数 mulMatrix 计算 m1 * m2
    mulMatrix(m1, m2, resultMul);
    //调用函数 diaplayMatrix 输出 m1 * m2 的结果 resultMul
    cout<<"                    ="<<endl;
    diaplayMatrix(resultMul);
    return 0;
}

/***************************************************************
Function: displayArray
Parameters:
        a       ----拟输出的矩阵
Description:
        将矩阵 a 输出
***************************************************************/
void diaplayMatrix( int a[][SIZE])
{
    for ( int indexLine=0; indexLine<=SIZE-1; indexLine++)
    {
        for ( int indexCol=0; indexCol<=SIZE-1; indexCol++)
            cout<<"\t"<<a[ indexLine ][ indexCol ];

        cout<<endl;
    }
}
```

```
/****************************************************************
Function: addMatrix
Parameters:
        a1              ----参加加法运算的第一个矩阵
        a2              ----参加加法运算的第二个矩阵
        result         ----保存加法结果的矩阵
Description:
        将矩阵 a1 和 a2 相加,结果保存在 result 中
****************************************************************/
void addMatrix( int a1[][SIZE], int a2[][SIZE], int result[][SIZE])
{
    for ( int indexLine=0; indexLine<=SIZE-1; indexLine++)
        for ( int indexCol=0; indexCol<=SIZE-1; indexCol++)
            result[indexLine][indexCol]=a1[indexLine][indexCol]+
                                        a2[indexLine][indexCol];

}
/****************************************************************
Function: mulMatrix
Parameters:
        a1              ----参加乘法运算的第一个矩阵
        a2              ----参加乘法运算的第二个矩阵
        result         ----保存乘法结果的矩阵
Description:
        将矩阵 a1 和 a2 相乘,结果保存在 result 中
****************************************************************/
void mulMatrix( int a1[][SIZE], int a2[][SIZE], int result[][SIZE])
{
    for ( int indexLine=0; indexLine<=SIZE-1; indexLine++)
        for ( int indexCol=0; indexCol<=SIZE-1; indexCol++)
            for ( int k=0; k<=SIZE-1; k++)
            result[indexLine][indexCol]+=a1[indexLine][k] *
                                          a2[k][indexCol];
}
```

程序执行结果:

```
*******************************************************************
1    1    1    1
2    2    2    2
3    3    3    3
4    4    4    4
        +
5    5    5    5
6    6    6    6
7    7    7    7
```

```
8     8     8     8
         =
6     6     6     6
8     8     8     8
10    10    10    10
12    12    12    12
**********************************************************************
1     1     1     1
2     2     2     2
3     3     3     3
4     4     4     4
        *
5     5     5     5
6     6     6     6
7     7     7     7
8     8     8     8
        =
26    26    26    26
52    52    52    52
78    78    78    78
104   104   104   104
```

6.3 字 符 串

6.3.1 字符与字符串概念

每个C++程序都是由一系列字符组合而成的。程序中可以包含字符常量,字符常量是用单引号括起来的一个字符,这个字符的值是机器字符集中该字符的整数值(如ASCII字符集中的值)。例如,'a'表示字a的整数值(ASCII字符集中为97),'\n'表示换行符的整数值(ASCII字符集中为10)。

在C++语言中,一个字符串就是用一对双引号括起来的一串字符,双引号是该字符串的起止标志符,它们不属于字符串本身。字符串可以包含字母、数字和+、-、*、/、$等各种特殊字符。例如:"string"、"Visual C++"、"a+b="、"姓名,年龄"和"Input a integer to x:"都是C++的字符串。

一个字符串的长度等于双引号内所有字符的长度之和,其中每个ASCII码字符的长度为1,而每个区位码字符(如汉字)按GB2312规定其长度为2。

特殊地,当一个字符串不含有任何字符时,则称为空串,用""表示,其长度为0;当只含有一个字符时,其长度为1,如"A"是一个长度为1的字符串。

注意:'A'和"A"是不同的,前者表示一个字符A,后者表示一个字符串A,虽然它们的值都是A,但它们具有不同的存储格式。

在一个字符串中不仅可以使用一般字符,而且可以使用转义字符。如字符串"\"cout

<<ch\"\n"中包含有 11 个字符,其中第 1 个和第 10 个为表示双引号的转义字符,最后一个为表示换行的转义字符。

6.3.2 字符串与字符数组

1. 字符串的存储

在 C++ 中,字符串的存储是利用一维字符数组来实现的,该字符数组的长度为待存字符串的长度加 1。设一个字符串的长度为 n,则用于存储该字符串的数组的长度应为 $n+1$。

把一个字符串存入数组时,是把每个字符依次存入数组的对应元素中,即把第一个字符存入下标为 0 的元素中,第二个字符存入下标为 1 的元素中,依次类推,最后把一个空字符'\0'存入下标为 n 的元素中,这里假定字符串的长度为 n。当然存储每个字符就是存储它的 ASCII 码或区位码。如利用一维字符数组 a[12]来存储字符串"Strings. \n"时,数组 a 中的内容如图 6-5 所示。

字符表示:

s	t	r	i	n	g	s	.	\n	\0		
0	1	2	3	4	5	6	7	8	9	10	11

ASCII码表示:

83	116	114	105	110	103	115	46	10	0		
0	1	2	3	4	5	6	7	8	9	10	11

图 6-5　存储字符串"strigns. \n"一维数组 a 中的内容

若一个数组被用来存储了一个字符串后,其尾部还有剩余的元素,最好也将其置为'\0'。

2. 利用字符串初始化字符数组

一个字符串能够在定义字符数组时作为初始化数据存入数组中,但不能通过赋值表达式直接赋值。例如:

```
(1) char a[10]= "array";
(2) char b[20]= "This is a pen. ";
(3) char c[8]= "";
(4) a= "struct";          //错误
(5) a[0]= 'A';
```

第(1)条语句定义了字符数组 a[10]并被初始化为"array",其中 a[0]～a[5]元素的值依次为字符'a'、'r'、'r'、'a'、'y'和'\0';第(2)条语句定义了字符数组 b[20],其中 b[i]元素(0≤i≤13)被初始化为所给字符串中的第 i+1 个字符,b[14]被初始化为字符串结束标志符'\0';第(3)条语句定义了一个字符数组 c[8]并初始化为一个空串,此时它的每个元素的值均为'\0';第(4)条语句是非法的,因为它试图使用赋值号把一个字符串直接赋值给一个数组,这在 C++ 中是不允许的;第(5)条是合法的,它把字符'A'赋给了 a[0]元素,使得数组 a 中保存的字符串变为"Array"。

利用字符串初始化字符数组也可以写成初值表的方式。例如,上述第(1)条语句与下面语句完全等效。

```
char a[10]={ 'a', 'r', 'r', 'a', 'y', '\0'};    //'\0'也可直接写为 0
```

注意：最后一个字符'\0'或 0 是必不可少的，它是一个字符串在数组中结束的标志。

3. 利用二维数组存储多个字符串

利用一维字符数组能够保存一个字符串，而利用二维字符数组能够同时保存若干个字符串，最多能保存的字符串个数等于该数组的行数。例如：

```
(1) char week[7][11]={"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"};
(2) char grade[][16]={ "excellent", "good", "middle", "pass", "bad"};
(3) char typeName[6][10]={ "int","double","char"};
(4) char d[10][20]={ "" };
```

在第(1)条语句中定义了一个二维字符数组 week，包含 7 行，每行有 11 个字符的空间，可以保存一个长度小于等于 10 的字符串，该语句同时对 week 进行了初始化，使得"Sunday"被保存到行下标为 0 的行里，该行包含 week [0][0]、week [0][1]、week [0][2]、week [0][3]、week [0][4]和 week [0][5] 6 个二维数组元素，每个元素的值依次为'S'、'u'、'n'、'd'、'a'和'y'，而 week [0][6]里存放了'\0'；同样"Monday"被保存到行下标为 1 的行里，"Tuesday"被保存到下标为 2 的行里，……"Saturday" 被保存到行下标为 6 的行里。以后既可以利用双下标变量 week [i][j](0≤i≤6，0≤j≤2)访问每个字符元素，也可以利用只带行下标的单下标变量 week [i](0≤i≤6)访问每行，它代表一个一维的字符数组，也可视为一个字符串。例如，week [2]表示字符串"Tuesday"，week[5]表示字符串"Friday"，参见图 6-6。

week										
S	u	n	d	a	y	\0	\0	\0	\0	\0
M	o	n	d	a	y	\0	\0	\0	\0	\0
T	u	e	s	d	a	y	\0	\0	\0	\0
W	e	d	n	e	s	d	a	y	\0	\0
T	h	u	r	s	d	a	y	\0	\0	\0
F	r	i	d	a	y	\0	\0	\0	\0	\0
S	a	t	u	r	d	a	y	\0	\0	\0

图 6-6　二维数组 week 存放示意图

上述第(2)条语句定义了一个二维字符数组 grade，它的行数没有显式地给出，取决于初始值的个数，因所列字符串为 5 个，所以该数组 grade 的行数为 5，又因列下标的上界定义为 16，所以每一行所存字符串的长度要小于等于 15。该语句被执行后，grade [0]表示字符串"excellent"，grade[1]表示字符串"good"……。

第(3)条语句定义了一个二维字符数组 typeName，它最多能够存储 6 个字符串，每个字符串的长度不能超过 9，该数组前 3 个字符串元素 typeName[0]、typeName[1]和 typeName[2]分别被初始化为"int"、"double"和"char"，后 3 个字符串元素均被初始化为空串。

第(4)条语句定义了一个能够存储 10 个字符串的二维字符数组 d，每个字符串的长度不能超过 19。该语句将所有数组元素初始化为'\0'。

6.3.3 字符串的输入和输出

用于存储字符串的字符数组，其元素可以通过下标运算符访问，这与一般字符数组和其他任何类型的数组都是相同的。例如，如果 a 是一个二维字符数组，则

```
cin>>a[4];
```

表示从键盘上向 a[4]输入一个字符串，而

```
cout<<a[i]
```

则表示向屏幕输出 a[i]中保存的字符串。

下面的程序段能够从键盘上依次输入 10 个字符串到二维字符数组 w 中保存起来，输入的每个字符串的长度不得超过 29。

```
#define N 10
char w[N][30];
for(int i=0; i<N; i++)
    cin>>w[i];
```

下面的一条 for 语句将按相反的次序依次输出在数组 w 中保存的所有字符串，在输出每个字符串之后都输出一个换行符。

```
for(i=N-1; i>=0; i--)
  cout<<w[i]<<endl;
```

除了上面通过下标运算访问字符数组外，C++ 还提供了对字符串进行整体输入输出的操作以及其他相关函数操作。如允许在提取或插入操作符后面使用一个字符数组名，假定 a 为一个一维字符数组，那么：

(1) cin>>a;

(2) cout<<a;

分别实现了向数组 a 中输入字符串或输出数组 a 中保存的字符串的功能。计算机执行上述第(1)条语句时，要求用户从键盘上输入一个字符串，以空格、制表符、回车符和文件结束符作为字符串输入的结束标志，系统就把该标志之前的字符串(不包括该标志)存入字符数组 a 中，当然在存入的整个字符串的后面将自动存入一个结束符'\0'。

注意：一定要保证输入的字符串的长度小于数组 a 的长度，这样才能把输入的字符串有效地存储起来，否则将引发越界，可能导致程序运行出错。另外，输入的字符串不需要另加双引号定界符，只要输入字符串本身即可，假如输入了双引号，则双引号将被视为一般字符。

执行上述第(2)条语句时，将向屏幕输出在数组 a 中保存的字符串，将从数组 a 中下标为 0 的元素开始，依次输出每个元素的值(实际上是按照元素的值输出其对应的字符)，直到碰到字符串结束符'\0'为止。若数组 a 中的内容如下所示(以字符形式描述，实际上存放的是对应的 ASCII 码)：

w	r	i	t	e	\0	R	e	a	d	\0
0	1	2	3	4	5	6	7	8	9	10

那么输出 a 时只会输出第一个空字符前面的字符串"write",而它后面的任何内容都不会被输出。同理,在其他情况下,以字符串方式来访问 a,都只认为其值是"write"。

利用插入操作符<<不仅能够输出字符数组中保存的字符串,而且能够直接输出一个字符串常量,即用双引号括起来的字符串,这种方式已经使用得非常多了。例如:

```
cout<< "x+ y= "<< x+ y<< endl;
```

此语句输出字符串"x+y="后接着输出 x+y 的值和一个换行符。若 x 和 y 的值分别为 15 和 24,则得到的输出结果为:

```
x+ y= 39
```

除了输入输出功能外,C++ 还提供了许多库函数用于对字符串进行操作,这些将在下一章中结合指针进行介绍。

习　题　6

6.1　回答问题:

(1) 什么是数组元素? 数组元素的下标有何特点?

(2) 什么是字符数组? 字符数组与字符串有何不同?

(3) 如何给数组元素赋初值?

6.2　编写完成下列任务的 C++ 语句:

(1) 显示字符数组 f 第 7 个元素的值。

(2) 将数值输入一维浮点数数组 b 的元素 4。

(3) 将一维整型数组 g 的 5 个元素各初始化为 5。

(4) 求出浮点数数组 d 的 100 个元素之和并打印这些元素。

(5) 将 a 数组复制到数组 b 的开头部分。假设有定义:

```
float a[11], b[40];
```

(6) 确定和打印 99 个元素的浮点数数组 w 中的最小值和最大值。

6.3　编写一个程序,实现输入 n 个数到一维数组,并求均方差:

$$D=\sum_{i=0}^{n-1}(a_i-M)^2,\quad 其中\ M=\sum_{i=0}^{n-1}a_i/n$$

6.4　用一个一维数组存放 1～10 的平方值,编写程序根据用户输入的数值(1～10)输出其平方。

6.5　定义一个可存放 30 个实数的一维数组来存放一个班(不超过 30 个人)某门功课的成绩。成绩由用户输入。设计一个循环过程,根据用户输入的号码(1～30)输出对应学生的成绩。当用户输入 0 时,程序结束。要求成绩的输入和输出用函数实现。

6.6　编写一个函数,该函数能够在一个已经按照从小到大排好序的整型数组中,快速找到某个指定的数据。请设计至少两种查找策略,并作比较。

6.7　有 15 个数存放在一个数组中,输入一个数,要求用二分查找法找出该数是数组中第几个元素的值。如果该数不在数组中,则打印出“无此数”。

6.8　设有一个数列,它的前 4 项为 0、0、2、5,以后每项分别是其前 4 项之和,编程求此数列的前 20 项。用一维数组完成此操作(提示:a[i]=a[i-1]+a[i-2]+a[i-3]+a[i-4];)。

6.9　用二维数组 int[20][2]存放一个班级的 20 名同学的物理和数学成绩，先设计一个函数输入同学的成绩，再设计一个函数求全班的总平均分和每一门课的平均分。

6.10　设计一个二维数组 float[30][6]，存放一个学生班（不超过 30 个人）中每人的 5 门功课的成绩及平均成绩。成绩由用户输入，平均成绩通过计算而得。设计一个循环过程，根据用户输入的号码（1～30）输出对应学生的各门成绩及平均成绩。要求成绩的输入和输出用函数实现。

6.11　设计一个程序，求一个 4 × 4 矩阵的两对角线元素之和。

6.12　编写求一个 3×5 阶矩阵与其自身转置矩阵的乘积的程序。

6.13　编写一个计算矩阵主负对角线元素和的函数。

6.14　“魔方”是指一个由正整数组成的方阵，它的每一行、每一列及每条主对角线上数据之和相等。例如，三阶魔方的一种形式如图 6-7 所示。

编写一个函数，函数的参数是整数 n，函数输出 $1 \sim n^2$ 的自然数组成的魔方。

6	1	8
7	5	3
2	9	4

图 6-7　三阶魔方

6.15　编写程序求出二维数组中的鞍点。所谓鞍点是指一个矩阵元素的值在其所在行中最大，在所在列中最小。

6.16　采用三维字符数组，输入、修改和显示一周课程表。

6.17　写一个函数，求字符串的长度。在 Main()函数中输入字符串，并输出其长度。

6.18　编写程序，能够输入一行字符，统计其中大写字母、小写字母和数字的个数。

6.19　已知某运动会上男子百米赛跑决赛成绩。要求编写程序，按成绩排序并按名次输出排序结果，包括输出名次、运动员号码和成绩三项内容。

提示：用 M 行 3 列数组存放运动员号码、成绩与名次，对决赛成绩降序排序，最后按排序后的位置输入名次。M 为运动员人数。

第 7 章 指 针

【学习内容】

本章介绍 C++ 的指针。主要内容包括:

◆ 指针的定义与运算。

◆ 指针与数组的关系。

◆ 字符串函数。

◆ 指针与 const 限定符。

◆ 传递指针参数。

◆ 动态内存分配方法。

◆ 函数指针。

【学习目标】

◆ 理解指针的概念。

◆ 掌握传递指针参数的机制。

◆ 理解指针、数组与字符串之间的关系。

◆ 掌握内存分配和释放的方法。

◆ 了解指针函数的作用。

无论是在 C 语言,还是在 C++ 中,指针都极其重要。利用指针,可以对内存中各种类型的数据进行快速、直接的处理,并为函数间的数据传递提供简洁、便利的方法。正确熟练地使用指针有利于编写高效的程序。当然,指针的使用也是非常复杂的,使用不当,特别是悬挂指针(指针没有指向任何对象)、内存泄漏(动态申请的内存空间没有正确释放)时,程序的性能会受到严重的影响,甚至导致严重错误。所以,指针很灵活,也很危险。

本章讨论指针的有关概念、指针的运算、指针与数组的关系,以及字符指针、函数指针、指针数组使用。

7.1 指针的定义

计算机中的数据都是存放在存储单元中,而每个存储单元都有一个地址,根据该地址可以定位存储单元,从而访问存储单元中的数据。一般变量的值存放在存储单元中,这些存储单元中存放的数据就是变量本身的值,可以通过变量名直接访问变量的值,这种访问称为直

接访问。如果把某个变量 i 对应的存储单元的地址(简称变量 i 的地址,假设为 01101100)放在另一个变量 p 对应的单元中,那么如果对变量 p 作直接访问,得到的是 p 所对应的单元内的数据——变量 i 的地址(01101100),如果取得这个地址后,按照该地址所指,再找到变量 i 所对应的单元内的数据——变量 i 的值为 8,这种通过 p 访问变量 i 的单元的访问方式就是间接访问,如图 7-1 所示。

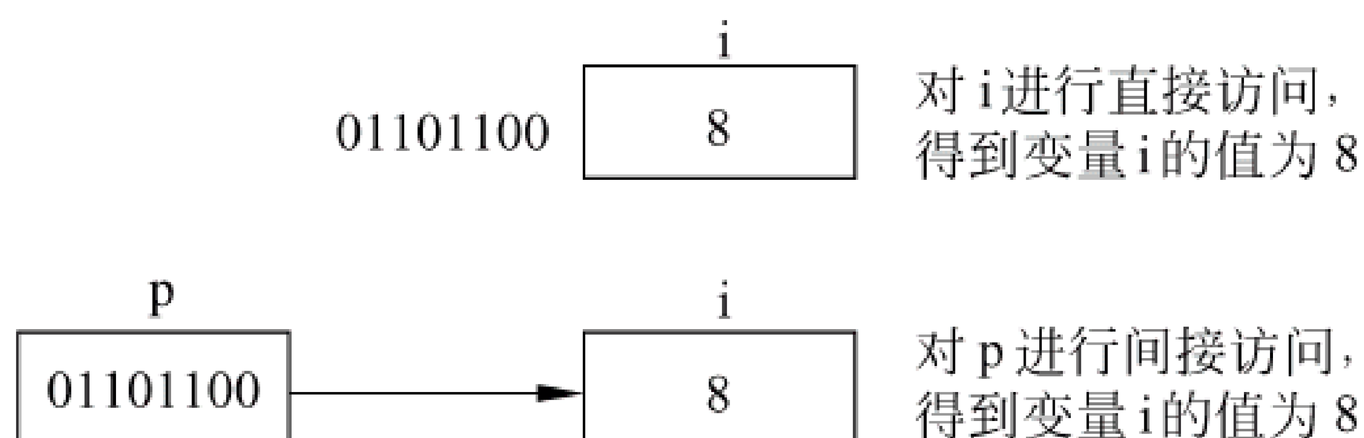


图 7-1　直接和间接访问

指针变量(简称指针)就是存放另一变量地址的变量,如图 7-1 中的变量 p。它与普通变量一样,也占用存储空间,也可以进行运算。指针与其他变量的不同之处在于,指针所对应的存储单元存放的是地址,而不是一般的数据。指针的概念类似于机器语言的间接寻址,在间接寻址方式中,一个存储单元中存放着另一个要处理数据的存储地址,这个存储单元就相当于指针。因此可以说,使用变量名可直接引用变量的值,而使用指针则可以通过间接访问的方式引用变量的值。

指针和其他变量一样,必须先声明后使用,其声明的形式与一般变量声明相比只是在变量名前多一个星号(*),例如:

```
int * p;
```

声明了变量 p 为指向整型值的指针(即变量 p 中存放的是某个整型变量的地址)。可以在一个语句中声明多个指针,也可以将指针和一般变量一起声明。

```
int * myPtr1, * myPtr2, i;
```

声明了两个指向整型值的指针 myPtr1 和 myPtr2 以及一个整型变量 i。指针可以指向任何数据类型的对象。又如,以下定义

```
float * xPtr, * yPtr;
```

表示 xPtr 和 yPtr 都是指向 float 值的指针。

指针可以在声明时进行初始化,也可以通过赋值语句获得值。指针可以取值为 0、NULL 或某个地址。具有值 NULL 的指针不指向任何值。NULL 是 C++ 预定义的符号常量,但它的值依赖于 C++ 的具体实现,有的版本将 NULL 定义为 0(如 Visual C++ 等),在这些版本中把一个指针初始化为 0 等价于把它初始化为 NULL,但是用 NULL 具有更好的可移植性。

7.2　指针的运算

与指针相关的运算有取地址运算、间接引用运算、算术运算、关系运算和赋值运算。

1. & 运算符

运算符 & 也称为“地址运算符”,是一元运算符,它返回操作数的地址。例如,假定有下

列语句

```
int y=5;
int * yPtr;
```

那么赋值语句

```
yPtr=&y;
```

是把变量 y 的地址赋给指针变量 yPtr,即让变量 yPtr 指向变量 y,如图 7-2 所示。

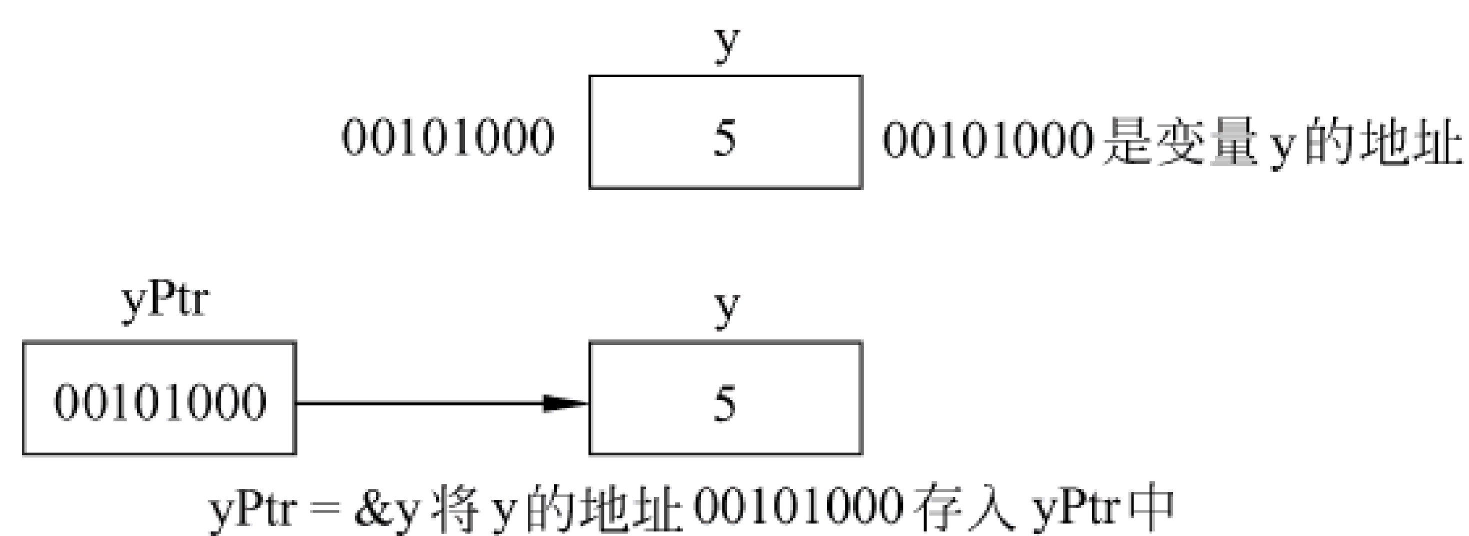


图 7-2　& 运算示意图

2. * 运算符

* 运算符通常称为“间接引用运算符”或“复引用运算符”,它返回其操作数(指针)所指向的对象。例如,如果接着上面的程序,执行语句

```
cout<< * yPtr<<endl;
```

将输出指针 yPtr 所指的整型变量 y 的值(即 5),在效果上如同下面的语句

```
cout<<y<<endl;
```

而语句

```
* yPtr=10;
```

将把 y 的值置成 10,在效果上如同下面的语句

```
y=10;
```

注意下面几点:

(1) * 的操作数不能是常量。

(2) * 和 & 的优先级相等,都与一元正(+)及一元负(−)相当。

(3) * 和 & 互为逆运算,如(≡表示“等价于”):

```
* &yPtr≡ * (&yPtr)≡ * (yPtr的地址) ≡ (yPtr的地址)指向的数据的值≡ yPtr
& * yPtr≡&(* yPtr)≡&(y)≡y的地址 (也是 yPtr的值)≡yPtr
```

所以 * &yPtr、& * yPtr 和 yPtr 都是等价的,因此 * 和 & 互为逆运算。

【例 7-1】 指针运算符 & 与 * 的示例。

```
//ex7_1:指针运算符 &与 *
# include <stdio.h>

int main()
```

```
{
    int n;                                   //n is an integer
    int * nPtr;                              //nPtr is a pointer to an integer
    n=7;
    nPtr=&n;                                 //nPtr set to address of n
    printf("The address of n is %p"
           "\nThe value of nPtr is %p", &n, nPtr);
    printf("\n\nThe value of n is %d"
           "\nThe value of * nPtr is %d", n, * nPtr);
    printf("\n\n& * nPtr=%p \n * &nPtr=%p\n", & * nPtr, * &nPtr);
    return 0;
}
```

程序执行结果：

```
The address of n is 0012FF7C
The value of nPtr is 0012FF7C
The value of n is 7
The value of * nPtr is 7
&* nPtr=0012FF7C
 * &nPtr=0012FF7C
```

printf 中的"%p"标志表示以预定义方式(通常是十六进制)显示一个指针的值,其对应的参数为 nPtr。程序执行结果显示,n 的地址和 nPtr 的值在输出中是一致的,这说明实际指针变量 nPtr 的值确实是变量 a 的地址。此外,& 和 * 运算符同时作用于 nPtr,输出结果均为 nPtr 的值,也说明 & 和 * 是互逆的。

3. 指针的算术运算

指针也可以参与算术运算。这时,运算对象是指针中存放的地址,指针运算本质上是地址的运算,因此指针的算术运算只有两种:加和减。指针作为操作数每加上或减去一个整数 n,其运算结果是指向指针当前指向的变量的后方或前方的第 n 个变量。假设 p1 和 p2 是指向相同类型的指针,n 为整数,指针可以进行的算术运算只有下面几种:

```
p1+n, p1-n, p1++, ++p1, p1--, --p1, p1-p2
```

尽管指针也是一种变量,由于它是对地址作运算,算术运算的语义与其他变量的算术运算有所不同。

(1) p1±n 是指将指针从当前位置向前或向后移动 n 个数据单位(每个数据单位字节大小等于指针指向的数据类型的对象的字节大小),而不是移动 n 个字节。由于指针可指向不同数据类型,即数据长度不同,所以此种运算的结果取决于指针指向的数据类型。假设它所指向的类型为 T,p1±n 的实际地址是:

```
p1±n* sizeof(T)
```

其中运算符 sizeof 返回其操作数类型 T 的数据占用的字节数目。假设指针 p 是指向整型值

的指针,且每个整数占4个字节,图7-3给出了p的若干算术运算的结果。

(2) p1－p2求出的是两指针位置之间的数据个数,而不是两指针持有的地址之差。p1－p2的结果按下列公式计算:

```
(p1的值-p2的值)/sizeof(T)
```

(3) p1++、++p1、p1--及--p1也是向前或向后移一个数据单位,不是移动一个字节。当它们与其他运算符用在一起时,应注意结合顺序。

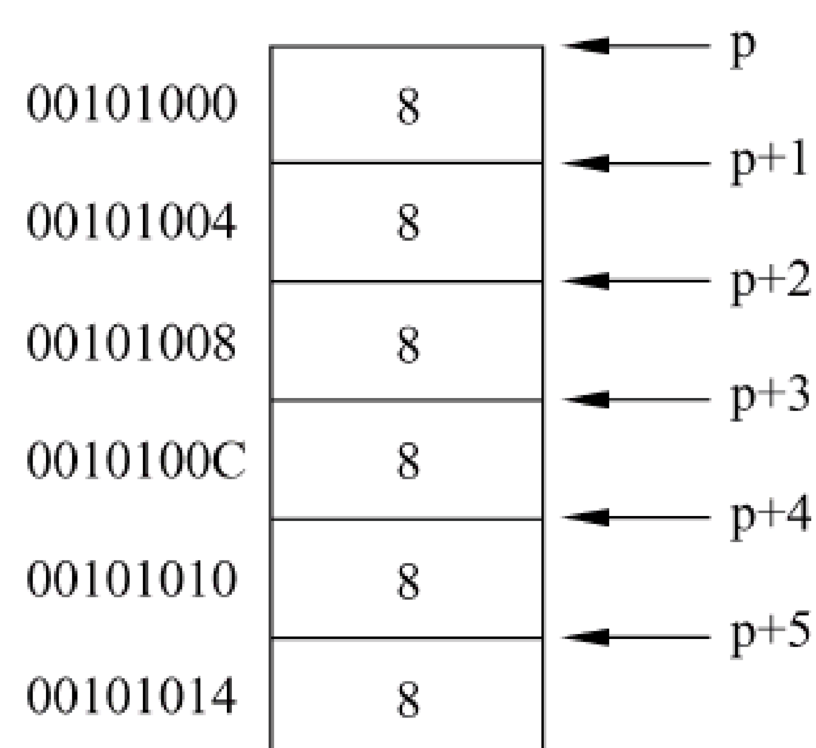


图7-3　指针的算术运算示意图

4. 指针的关系运算

在关系表达式中可对相同类型的两个指针进行各种关系运算,其结果可以反映两指针所指向的地址之间的位置前后关系。例如,对于两指针p1和p2(它们指向相同的数据类型),关系表达式:

```
p1<p2
```

当p1所指位置在p2之前时,该表达式的值为true,否则为false。

指针也可以和0或NULL进行==或!=的关系运算。例如:

```
p1==0或 p1!=0
```

可以用它们来判断p1是否为空指针。除此之外,不同类型指针之间以及指针和一般整数间进行的关系运算是没有意义的。

5. 指针的赋值运算

为指针变量赋值时,赋的值必须是地址常量或变量,而不能是普通整数,它有下面几种形式。

(1) 把变量地址赋给指向相同数据类型的指针。

```
char c, * pc ;
pc=&c;
```

(2) 把指针赋给相同数据类型的另一个指针。

```
int * p, * q;
p=q;
```

(3) 把数组的地址(即数组变量的值,即数组第一个元素的地址)赋给指向相同数据类型的指针。

```
char name[20], * pName;
pName=name;                          //name的值就是数组第一个元素的地址,即数组的地址
```

(4) 此外,还可使用下列形式的赋值。

```
int * p, * q, n;
p=q+n;
p=q-n;
```

```
p+=n;
P-=n;
```

【例 7-2】 指针的算术运算、关系运算和赋值运算示例。

```
//ex7_2: 指针的算术运算、关系运算和赋值运算
#include<stdio.h>

int main()
{
    char c='A', *pc;

    int  ints[5]={1,2,3,4,5}, *p1;
    int  m=6, n=7;
    int *p2, *p3;

    pc=&c;
    printf("c=\'%c\' \t pc=%p \t *pc=\'%c\' \n", c, pc, *pc);
    pc+=5;
    printf("After pc+=5: \t pc=%p\n", pc);

    p1=ints;                     //ints 的值就是数组第一个元素的地址,即数组的地址
      printf("\np1=%p \t *p1=%d \n", p1, *p1);
    p1+=4;
      printf("After p1+=4:\t p1=%p \t *p1=%d \n", p1, *p1);

    p2=&m;
    p3=&n;
    printf("\np2=%p \t *p2=%d \n", p2, *p2);
    printf("p3=%p \t *p3=%d \n", p3, *p3);
    printf("p2-p3=%d \n", p2-p3);

    if (p2<p3)
        printf("p2<p3 \n");
    else
        printf("p2>=p3 \n");

    return 0;
}
```

程序执行结果：

```
c='A'    pc=0012FF7C    *pc='A'
After pc+=5:    pc=0012FF81

p1=0012FF64      *p1=1
After p1+=4:    p1=0012FF74    *p1=5

p2=0012FF5C      *p2=6
p3=0012FF58      *p3=7
p2-p3=1
p2>=p3
```

7.3 指针与数组

在上节中,已经注意到,可以将数组赋值给指针,这是因为数组名(数组变量)的值就是数组第一个元素的地址,因此可以将数组的值赋给指针,使得指针指向数组的第一个元素。当然,这有一个前提,即数组元素的类型和指针指向的类型是相同的。可见,在C++中,数组和指针有着密切的关系,二者甚至可以互换使用:数组名可以视为一个指针,只是这个指针的值在数组声明后就不能变化,这种指针称为常量指针。通过指针也可以访问数组的每个元素。此外,数组元素也可以是指针。

1. 指针和数组互换使用

在C++中,由于指针与数组变量都表示地址量,因此可以互换使用。数组名可以看成常量指针,也可以用指针代替数组名来访问数组中的元素。例如,下列语句

```
int a[10], * aPtr;
```

定义了整型数组a[10]和整数指针变量aPtr。由于数组名(不带下标)是数组的第一个元素的指针,因此可以用下列语句将aPtr的值设置为数组a第一个元素的地址。

```
aPtr= a;
```

这相当于取数组第一个元素的地址,如同下面的语句

```
aPtr= &a[0];
```

数组元素a[5]也可以用指针表达式来引用。例如:

```
* (aPtr+ 5)
```

表达式中的5是指针的偏移量,(aPtr+5)指向了下标为5的数组元素(5代表5个元素,而不是5个字节)。可见,如果通过指针访问数组元素,当指针指向数组首地址时,偏移量表示要引用的数组元素,偏移量的值等于数组下标。表达式中括号是必须加的,因为*的优先级高于+的优先级。

就像数组元素可以通过指针表达式引用一样,下面的表达式

```
&a[5]
```

等价于指针表达式

```
aPtr+ 5
```

数组名(数组变量)本身可以当作指针并参加指针运算。例如,表达式

```
* (a+ 5)
```

同样引用数组元素a[5]。注意,上述表达式不修改a的值,a还是指向数组的第一个元素。

指针和数组一样可以加下标。例如,表达式

```
aPtr[5]
```

同样指向数组元素a[5]。

【例 7-3】 数组和指针的互换性示例。

```
//ex7_3: 数组和指针的互换性示例
#include<stdio.h>
#define SIZE 5
int main()
{
    int a[SIZE]={ 0, 1, 2, 3, 4 };
    int *aPtr=a;                          //aPtr指向数组 a
    int i, offset;
    //通过数组名和下标访问数组
    printf("Access array through array and subscript: \n");
    for (i=0; i<SIZE; i++)
        printf("a[%d]=%d\n", i, a[i]);
    //通过指针和偏移量访问数组
    printf("\nAccess array through pointer and offset: \n");
    for (offset=0; offset<SIZE; offset++)
        printf("*(aPtr+%d)=%d\n", offset, *(aPtr+offset));
    //通过指针和下标访问数组
    printf("\nAccess array through pointer and subscript: \n");
    for (i=0; i<SIZE; i++)
        printf("aPtr[%d]=%d\n", i, aPtr[i]);
    //通过数组名和偏移量访问数组
    printf("\nAccess array through array and offset: \n");
    for (offset=0; offset<SIZE; offset++)
        printf("*(a+%d)=%d\n", offset, *(a+offset));
    return 0;
}
```

程序执行结果：

```
Access array through array and subscript:
a[0]=0
a[1]=1
a[2]=2
a[3]=3
a[4]=4
Access array through pointer and offset:
*(aPtr+0)=0
*(aPtr+1)=1
*(aPtr+2)=2
*(aPtr+3)=3
*(aPtr+4)=4
Access array through pointer and subscript:
```

```
aPtr[0]=0
aPtr[1]=1
aPtr[2]=2
aPtr[3]=3
aPtr[4]=4

Access array through array and offset:
*(a+0)=0
*(a+1)=1
*(a+2)=2
*(a+3)=3
*(a+4)=4
```

从这个例子可以看出,指针和数组能够互换使用,指针的运算形式和数组下标形式是等价的。一般通过指针访问数组元素比通过下标访问数组元素更加直接和高效,因为数组下标在编译时还是要变成指针表示法,但是使用数组下标要更直观和清楚。另外,在访问数组元素时,二者都应注意,不要超过下标范围。

尽管指针和数组下标在访问数组时可互相替代,但指针与数组还有差别:指针是地址变量,在程序中可改变其值;而数组名是地址常量,是个其值不可改变的常量指针。

2. 指针数组

由于指针也是一种变量,也可以是数组的元素,即数组可以包含指针。这种数据结构通常用来构造字符串数组,即元素是字符串的数组。字符串数组中每一项都是字符串,在C++中,字符串是存储在一个一维字符数组中,字符串也可以视为一个指向字符的指针,该指针指向了字符串中第一个字符。所以字符串数组是一种指针数组,每一个元素都是指向字符的指针。例如,下列语句:

```
char * week[7]={"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday"};
```

定义的week[7]表示7个元素的数组,数组的每个元素都是char * 类型,即指向char类型的指针。数组的7个值为"Sunday"、"Monday"、"Tuesday"、"Wednesday"、"Thursday"、"Friday" 和 "Saturday"。每个值存放在内存中的某个连续区域中,每个字符串的结尾都增加了一个终止字符'\0',因而每个字符串实际占用的字节数分别为7、7、8、10、9、7、9。尽管这些字符串好像是放在week数组中,其实数组中存放的只是指针,如图7-4所示。每个指针指向对应字符串中的第一个字符。对比6.3.2节定义的二维数组:

```
char week[7][11]={"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
"Friday","Saturday"};
```

的存储方式(见图6-6),可以发现,采用二维数组存放字符串,需要按照数组中最长的字符串来声明数组,这将会造成空间浪费;如果采用指针数组来存储字符串数组,尽管week数组是固定长度的,但可以存放任意长度的字符串,而不会造成空间的浪费。这是C++强大的数据结构功能所带来的灵活性。

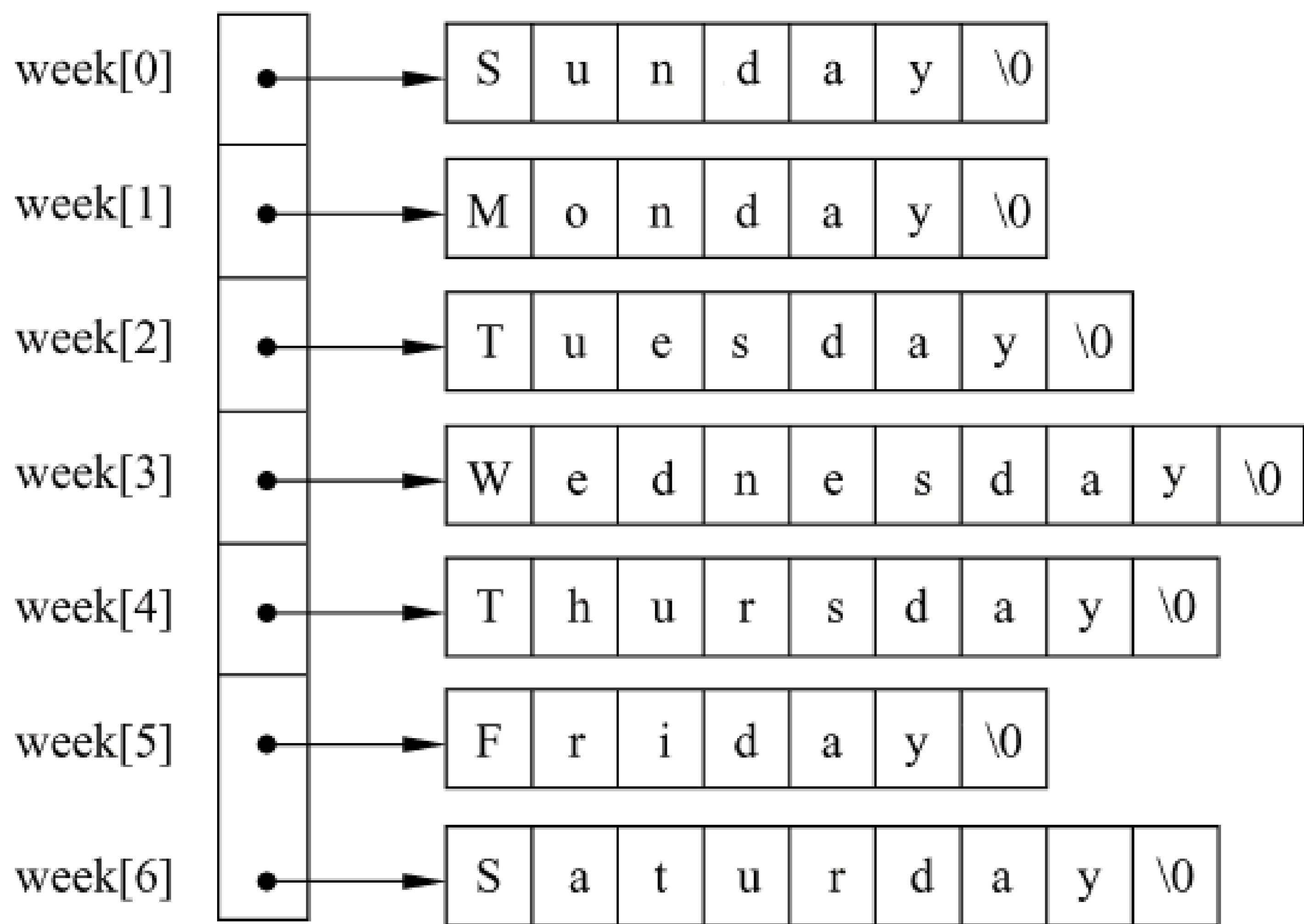


图 7-4　指针数组 week 示意图

7.4　字符指针与字符数组

可以看到,既可以用字符数组来表示字符串,也可以用字符指针来表示它,只要将字符指针指向字符串的首地址就可以了。这是因为字符串是连续存放的,并且以'\0'结尾,确定了字符串的首地址,也就确定了整个字符串。例如:

```
char * s="string1";
cout<< s<< endl;
s="string2";
cout<< s<< endl;
```

可以看出,将字符串赋给字符指针是非常直接和方便的。将字符串赋给字符指针时,不是将字符串复制到指针中,而是把字符串的首址赋予指针。上面的程序中,编译器会为字符串常量"string1"和"string2" 分配空间,将其存放到内存中某处。初始化时,首先将字符串"string1"的首地址赋予指针 s,所以第一个输出语句在屏幕上显示 string1,接下来的赋值语句将"string2"的首地址赋予指针 s,第二个输出语句将输出 s 指向的字符串"string2"。

C++ 为字符串的操作提供了丰富的函数,如连接、比较、复制等。下面对常用的字符串处理函数作简单介绍。

7.4.1　字符串函数

C++ 提供的处理字符串的预定义函数原型在头文件 string.h 中声明,要使用这些函数时,必须在程序文件中使用#include 命令将该头文件包括进来。

C++ 系统提供的处理字符串的预定义函数有很多,从 C++ 库函数资料中可以得到全部说明。表 7-1 简要列出了这些函数。

表 7-1　常用的字符串处理函数

函数原型	函数功能
char * strcpy(char * dest, const char * src)	将字符串 src 复制到 dest
char * strcat(char * dest, const char * src)	将字符串 src 添加到 dest 末尾
char * strchr(const char * s, int c)	检索并返回字符 c 在字符串 s 中第一次出现的位置
int strcmp(const char * s1, const char * s2)	比较字符串 s1 与 s2 的大小,若 s1 串大于 s2 串则返回一个大于 0 的值;若 s1 串等于 s2 串则返回值为 0;若 s1 串小于 s2 串则返回一个小于 0 的值
size_t strcspn(const char * s1, const char * s2)	扫描 s1,返回在 s1 中和 s2 中都出现的字符的个数
char strdup(const char * s)	将字符串 s 复制到最近建立的单元
int stricmp(const char * s1, const char * s2)	按照字母的小写比较字符串 s1 和 s2,若 s1 串大于 s2 串则返回一个大于 0 的值;若 s1 串等于 s2 串则返回值为 0;若 s1 串小于 s2 串则返回一个小于 0 的值
size_t strlen(const char * s)	返回字符串 s 的长度
char * strlwr(char * s)	将字符串 s 中的大写字母全部转换成小写字母,并返回转换后的字符串
char * strncat(char * dest, const char * src, size_t n)	将字符串 src 中最多 n 个字符复制到字符串 dest 中
int strncmp(const char * s1, const char * s2, size_t n)	比较字符串 s1 与 s2 中的前 n 个字符
char * strncpy(char * dest, const char * src, size_t n)	复制 src 中的前 n 个字符到 dest 中
int strnicmp(const char * s1,const char * s2, size_t n)	比较字符串 s1 与 s2 中的前 n 个字符
char * strnset(char * s,int ch, size_t n)	将字符串 s 的前 n 个字符置于 ch 中
char * strpbrk(const char * s1, const char * s2)	扫描字符串 s1,并返回在 s1 和 s2 中都出现的字符个数
char * strrchr(const char * s, int c)	扫描最后出现一个给定字符 c 的一个字符串 s
char * strrev(char * s)	将字符串 s 中的字符全部颠倒顺序重新排列,并返回排列后的字符串
char * strset(char * s, int ch)	将一个字符串 s 中的所有字符置于一个给定的字符 ch
size_t strspn(const char * s1, const char * s2)	扫描字符串 s1,并返回在 s1 和 s2 中均有的字符个数
char * strstr(const char * s1, const char * s2)	扫描字符串 s2,并返回第一次出现 s1 的位置
char * strtok(char * s1, const char * s2)	检索字符串 s1,该字符串 s1 是由字符串 s2 中定义的定界符所分隔
char * strupr(char * s)	字符串 s 中的小写字母全部转换成大写字母,并返回转换后的字符串

下面简要介绍其中几个主要的字符串函数的使用。

1. 求字符串长度

函数原型:

```
int strlen(const char * s);
```

strlen 函数只有一个参数，该参数是一个指向字符的指针，该指针代表了一个字符串，它前面使用的保留字 const 表示该字符串的内容在函数体中不允许被修改，只能读取。调用该函数时，对应的实参可以为任何形式的字符串，如可以是一个字符串常量，可以是一个一维字符数组变量，也可以是二维字符数组中只带行下标的单下标变量，当然也可以是一个字符指针。该函数将返回实参字符串的长度。

例如，假设有 3 个字符数组 a[10]、b[10]和 c[20]，它们的内容分别是""、"b"和"Hello world"，则 strlen(a)、strlen(b)和 strlen(c)的值分别为 0、1 和 11。直接使用 strlen("Hello world")也可得到字符串的长度 11。

2. 字符串复制

函数原型：

```
char * strcpy(char * dest, const char * src);
```

strcpy 函数有两个参数，都是字符指针。因为每个字符指针是指向相应字符串的首地址，字符数组名也可视为一个字符指针，所以上述函数原型与

```
char * strcpy(char dest[], const char src[]);
```

是等价的。

strcpy 函数的功能是把第二个参数 src 所指字符串(包括结束标志'\0')复制到第一个参数 dest 所指的存储空间中，然后返回 dest 的值，即一个字符指针，它指向的区域中存放了复制好的字符串。显然，dest 对应的实参应该是一个字符数组变量，或者是一个地址，该地址指向的连续空间用来存放复制的字符串。注意，使用该函数时一定要保证 dest 对应的实参提供的存储空间足够存放被复制的字符串。此外，dest 对应的实参不能是字符串常量，尽管字符串常量满足参数类型的要求，但是系统为字符串常量分配的空间是不允许修改的，试图将被复制的字符串往字符串常量的空间中复制时，会导致非法访问的运行错误。需要特别指出的是，如果 src 和 dest 所指的存储空间有重叠，那么该函数的执行结果不确定。

因为 strcpy 函数只需要从 src 字符串中读取内容，不需要修改它，所以用 const 修饰，而对于第一个参数 dest，需要修改它的内容，所以就不能用 const 修饰。

分析下面的程序段：

```
char a[10], b[10]="copy";
strcpy(a, b);
cout<< "a= \" "<< a << "\" b= \" "<< b << "\"<< endl;
cout<< "strlen(a)= "<< strlen(a)<< "strlen(b)= "<< strlen(b)<< endl;
```

上述程序段首先声明了两个字符数组 a 和 b，并对 b 初始化为"copy"；接着调用 strcpy 函数，把 b 所指向(即数组 b 保存)的字符串"copy"复制到 a 所指向(即数组 a 占用)的存储空间中，使得数组 a 保存的字符串同样为"copy"；第三条语句输出 a 和 b 所指向的字符串，或者输出数组 a 和 b 中所保存的字符串；第四条语句输出 a 和 b 所指向的字符串的长度。该程序段的运行结果为：

```
a= "copy" b= "copy"
strlen(a)= 4 strlen(b)= 4
```

字符串复制还有另外一个类似的函数:

```
char * strncpy(char * dest, const char * src, size_t n);
```

strncpy 函数与 strcpy 类似,把第二个参数 src 所指字符串中的前 n 个字符复制到第一个参数 dest 所指的存储空间中(如果 src 所指字符串的长度小于 n,则复制整个字符串),然后返回 dest 的值,其中 size_t 是无符号整数类型。

3. 字符串连接

函数原型:

```
char * strcat(char * dest, const char * src);
```

strcat 函数同 strcpy 函数具有完全相同的参数说明和返回值类型。函数功能是把第二个参数 src 所指字符串添加到第一个参数 dest 所指字符串之后,即 src 所指字符串复制到 dest 所指字符串之后的存储空间中。该函数返回 dest 的值。需要特别指出,如果 src 和 dest 所指的存储空间有重叠,那么该函数的执行结果也不确定。

使用 strcat 函数时要确保 dest 所指字符串之后有足够的存储空间用于存储 src 所指的字符串,否则会导致访问越界的错误。

调用 strcat 函数之后,第一个实参所指字符串的长度将等于两个实参所指字符串的长度之和。例如:

```
char a[20]="string";                        //字符串长度为 6
char b[]="catenation";                      //字符串长度为 10
strcat(a, " ");                             //连接一个空格到 a 串之后
strcat(a, b);                               //把 b 串连接到 a 串之后
cout<<a<<endl<<"strlen(a)="<<strlen(a)<<endl;
```

执行该程序段得到的输出结果为:

```
string catenation
strlen(a)=17
```

字符串连接也有另外一个类似的函数:

```
char * strncat(char * dest, const char * src, size_t n);
```

strncat 函数与 strcat 类似,把第二个参数 src 所指字符串中的前 n 个字符添加到第一个参数 dest 所指的字符串的后面(如果 src 所指字符串的长度小于 n,则添加整个字符串),然后返回 dest 的值。

4. 字符串比较

函数原型:

```
int strcmp(const char * s1, const char * s2);
```

strcmp 函数带有两个字符指针参数,分别指向进行比较的字符串,函数的返回值为整型。其功能为:比较 s1 所指字符串与 s2 所指字符串的大小,若 s1 串大于 s2 串则返回一个大于 0 的值;若 s1 串等于 s2 串则返回值为 0;若 s1 串小于 s2 串则返回一个小于 0 的值。

比较 s1 串和 s2 串的大小是一个循环过程，从两个串的第一个字符起从前往后依次比较它们的 ASCII 码值，直到遇到被比较的两个字符不同或到某个字符串到达末尾为止，整个比较过程可用下面的程序段描述出来。

```
int i;
for(i=0; s1[i] && s2[i]; i++)
    if (s1[i] > s2[i]) return 1;
    else if (s1[i]< s2[i]) return -1;
if (s1[i] ==0 && s2[i] ==0) return 0;
else if (s1[i] !=0) return 1;
else return -1;
```

在这个程序段中使用 s1[i]和 s2[i]来表示 s1 数组和 s2 数组中下标为 i 的元素，即 s1 和 s2 所指字符串中的第 i+1 个字符。

例如，下面的关系是成立的：

```
strcmp("1234abc", "1234") >0
strcmp("1234abc", "1234abd")<0
strcmp("1234abc", "1230") >0
strcmp("A","a")<0
```

同样，strcmp 也有一个类似的函数：

```
char * strncmp(const char * s1, const char * s2, size_t n);
```

strncmp 函数与 strcmp 类似，比较 s1 所指字符串与 s2 所指字符串的前 n 个字符的大小(如果 s1 或 s2 所指字符串的长度小于 n，则比较到 s1 和 s2 任意一个字符串的结尾处就结束)，返回结果同 strcmp。

5. 从字符串中查找字符

函数原型：

```
char * strchr(const char * s, int c);
```

strchr 函数查找某个字符在字符串中第一次出现的位置，即从 s 所指字符串中的第一个字符起，顺序查找 ASCII 码为 c 的字符的首次出现的位置，若查找成功则返回该字符的存储地址，否则返回 NULL。

当调用 strchr 函数时，与形参 c 对应的实参可以为整数，但通常是一个待查找的字符。

例如，函数调用 strchr("abcd",'c')将返回字符串"abcd"的首地址加 2 的值，而函数调用 strchr("abcd",'e')将返回 NULL。

6. 从字符串中逆序查找字符

函数原型：

```
char * strrchr(const char * s, int c);
```

strrchr 与 strchr 函数功能类似，都是从字符串中查找字符，但该函数是从 s 所指字符串的最后一个字符起顺序向前查找 ASCII 码为 c 值的字符的首次出现的位置，若查找成功则返回字符的存储地址，否则返回 NULL。实际上，该函数查找的是某个字符在字符串中

最后一次出现的位置。

例如,函数调用strrchr("abcab",'a')将返回字符串"abcab"的首地址加3的值,若把函数名改为strchr,则返回结果为"abcab"的首地址值。

7. 从字符串中查找子串

函数原型:

```
char * strstr(const char * s1, const char * s2);
```

strstr函数从s1所指字符串中第一个字符起,顺序向后查找出与s2所指字符串相同的子串,若查找成功则返回该子串首次出现的首地址,否则返回NULL。例如:

```
char a[20]="abcdeabcde";
char b[4]="bcd";
char c[4]="dec";
cout<<strstr(a,b)<<endl;
if (strstr(a, c)==NULL)
    cout<<"Not found!"<<endl;
```

该程序段首先定义3个字符数组并分别进行初始化,接着输出以strstr(a, b)的返回地址为首地址的字符串"bcdeabcde",执行if语句时,因从a串中查找不到c串,所以判断表达式成立,将向屏幕输出字符串"Not found!"和一个换行符。

8. 字符串分解

函数原型:

```
char * strtok(char * s, char * delim);
```

strtok函数的功能是,将字符串分解为一组标记串。其中,s为要分解的字符串,delim是由作为分隔符的字符组成的串,即以delim中出现的字符为分隔标志,将s分解成若干个子串。函数strtok的使用有些特殊,它采取了一种反复调用的方式,每次调用时,strtok在s中查找包含在delim中的字符并用'\0'来替换,返回指向的当前标记,直到找遍整个字符串。当没有标记串时则返回空字符NULL。首次调用时,第一参数s必须指向要分解的字符串,后续调用时s对应的实参必须为NULL。例如:

```
char s[]="Hello C++ World";
char * d=" ";
char * p;
p=strtok(s, d);
while(p)
{
    printf("%s\n", p);
    p=strtok(NULL, d);
}
```

该程序段将字符串"Hello C++ World"以空白符为分隔符进行分解,并将分解的每个子串(单词)输出:

```
Hello
```

```
C++
World
```

7.4.2　字符串函数的应用

本节通过具体的例子进一步理解字符串函数的作用,正确掌握它们的使用方法。

【例 7-4】 字符串函数 strcpy 和 strncpy 的使用示例。

```
//ex7_4.cpp: 字符串函数 strcpy 和 strncpy 的使用
#include<iostream.h>
#include<string.h>
int main()
{
    char s1[20]="Hello C++  World!";
    char s2[20]="Hello New World!";
    char s3[20]="";
    cout<<"s1=\""<<s1<<"\""<<endl;
    cout<<"s2=\""<<s2<<"\""<<endl;
    cout<<"s3=\""<<s3<<"\""<<endl;
    cout<<"strcpy(s3, s1)=\""<<strcpy(s3, s1)<<"\""<<endl ;
    cout<<"strncpy(s3, s2, 9)=\""<<strncpy(s3, s2, 9)<<"\""<<endl;
    return 0;
}
```

程序执行结果:

```
s1="Hello C++ World!"
s2="Hello New World!"
s3=""
strcpy(s3, s1)="Hello C++ World!"
strncpy(s3, s2, 9)="Hello New World!"
```

程序执行完 strcpy(s3, s1)后,s3 的内容为"Hello C++ World!",而后执行 strncpy(s3, s2, 9),将 s2 的前 9 个字符"Hello New"复制到 s3 中,因而覆盖了 s3 的前 9 个字符。注意,这次操作并未在复制完 s2 的前 9 个字符后在 s3 中紧接着添加'\0',通过字符指针 s3 来访问其所指的字符串时,字符串的结束标志还是原来在'!'之后的'\0'。因此,输出结果为"Hello New World!"。

【例 7-5】 字符串函数 strcat 和 strncat 的使用示例。

```
//ex7_5.cpp: 字符串函数 strcat 和 strncat 的使用
#include<iostream.h>
#include<string.h>
int main()
{
    char s1[20]="Hello ";
```

```
    char s2[10]="C++ World";
    char s3[]="World!";
    cout<<"s1=\""<<s1<<"\""<<endl;
    cout<<"s2=\""<<s2<<"\""<<endl;
    cout<<"s3=\""<<s3<<"\""<<endl;
    cout<<"strncat(s1, s2, 4)=\""<<strncat(s1, s2, 4)<<"\""<<endl;
    cout<<"strcat(s1, s3)=\""<<strcat(s1, s3)<<"\""<<endl;
    return 0;
}
```

程序执行结果:

```
s1="Hello "
s2="C++ World"
s3="World!"
strncat(s1, s2, 4)="Hello C++ "
strcat(s1, s3)="Hello C++ World!"
```

【例 7-6】 字符串函数 strcmp 和 strncmp 的使用示例。

```
//ex7_6.cpp:字符串函数 strcmp 和 strncmp 的使用
#include<iostream.h>
#include<string.h>
int main()
{
    const char * s1="Hello C++ World!";
    const char * s2="Hello C++ World!";
    const char * s3="Hello World!";
    cout<<"s1=\""<<s1<<"\""<<endl;
    cout<<"s2=\""<<s2<<"\""<<endl;
    cout<<"s3=\""<<s3<<"\""<<endl;
    cout<<"strcmp(s1, s2)="<<strcmp(s1, s2)<<endl;
    cout<<"strcmp(s1, s3)="<<strcmp(s1, s3)<<endl;
    cout<<"strncmp(s1, s2, 6)="<<strncmp(s1, s2, 6)<<endl;
    cout<<"strncmp(s1, s2, 7)="<<strncmp(s1, s2, 7)<<endl;
    cout<<"strncmp(s1, s3, 7)="<<strncmp(s1, s3, 7)<<endl;
    cout<<"strncmp(s3, s2, 7)="<<strncmp(s3, s2, 7)<<endl;
    return 0;
}
```

程序执行结果:

```
s1="Hello C++ World!"
s2="Hello C++ World!"
s3="Hello World!"
```

```
strcmp(s1, s2)=0
strcmp(s1, s3)=-1
strncmp(s1, s2, 6)=0
strncmp(s1, s2, 7)=0
strncmp(s1, s3, 7)=-1
strncmp(s3, s2, 7)=1
```

【例 7-7】 字符串函数 strchr、strstr 和 strtok 的使用示例。

```
//ex7_7.cpp: 字符串函数 strchr、strstr 和 strtok 的使用
#include<iostream.h>
#include<string.h>
int main()
{
    char sent[]="Hello C++ World!I am coming!";
    char * delim=" !";                           //空格和!是分隔符
    char * token;
    cout<<"sent=\""<<sent<<"\""<<endl;
    //使用 strchr 和 strstr 进行查找
    cout<<"\nThe string in sent following first \'o\' is :";
    cout<<strchr(sent, 'o')<<endl;
    cout<<"The string in sent following \"C++\" is :";
    cout<<strstr(sent, "C++")<<endl;
    //使用 strtok 将 sent 分解成一个个标记(即单词)
    cout<<"\nThe sentence is splited to:"<<endl;
    token=strtok(sent, delim);
    while(token)
    {
        cout<<token<<endl;
        token=strtok(NULL, delim);
    }
    return 0;
}
```

程序执行结果:

```
sent="Hello C++ World!I am coming!"

The string in sent following first 'o' is :o C++ World!I am coming!
The string in sent following "C++" is :C++ World!I am coming!

The sentence is splited to:
Hello
C++
World
I
am
```

```
coming
```

从上面的程序可以看出,每次调用 strtok,都会导致被分解的字符串中当前标记后面的那个分隔符被替换成'\0',所以 strtok 会破坏被分解的字符串,如果还需要使用该字符串,则在调用该函数之前要先将字符串复制一个备份。

7.5 指针与 const 限定符

7.5.1 基本概念

在上一节中,已经看到了通过使用 const 限定符可以明确地说明哪些数据是不会改变的。const 限定符通常在声明变量时使用。例如:

```
const int studNum=100;
```

说明 studNum 是一个整型变量,初始值为 100,而且 studNum 是不可再修改的。实际上,使用 const 限定符后,studNum 相当于一个常量名,程序中任何试图修改 studNum 的代码都会被编译器诊断出来。声明常量数据时都要给出初始值。

在 C++ 中,const 限定符告诉编译器其限定的数据对象是不能修改的,编译器一旦发现有程序代码试图修改这些数据,就会报告错误,提示程序员。通过这一手段,可以使程序员避免因为无意识地修改某些数据而带来的错误。指针为 C++ 提供了非常自由的处理数据的手段,这种方便的措施有时候也会带来负面的影响。例如,在编写大规模的软件时,很难控制指针的安全使用,易造成指针的访问错误。如果结合 const 限定符则可以在一定程度上避免这类错误。

软件工程提示: *软件工程中有一个最低权限原则,即将对数据的访问权限和范围限制在尽可能小的程度内,这样可以尽量利用编译器来发现程序中潜在的错误。尽可能少用全局变量,多使用 const 限定符都是这一原则的体现。*

*7.5.2 用 const 限定指针

指针作为一种变量,它也可以和 const 限定符结合起来,限制对指针变量和指针所指对象的修改。const 限定符在指针中的使用是非常灵活的。

1. 指向非常量的非常量指针

如果声明指针时不使用 const,就可以定义指向非常量的非常量指针。

```
int val1, val2;
int * p;
```

上述代码声明了两个整型变量 val1 和 val2,以及一个指向整数的指针 p。由于都未使用 const 进行限制,所以直接对它们的修改均是合法的。例如:

```
val1=10;
val2=20;
p=&val1;
 * p+=10;
```

```
p=&val2;
*p+=10;
```

没有使用 const 确实给了程序操作数据的最大权限,可以任意修改变量,这样做固然为程序设计带来了方便,但更有可能带来了程序安全性的隐患,需要将程序操作数据的权限限定在最恰当的范围内。const 可以支持这种限定。

2. 指向常量数据的非常量指针

可以使用 const 来限定指针指向的数据。考虑下面的声明:

```
const int *p;
```

说明 p 是一个指向常整数的指针,即 p 指向的整数是不能通过 p 来修改的,即不能通过 *p 来修改该整数,但这并不意味着该整数是不可修改的,实际上可以通过其他途径来修改。考虑下面的程序段:

```
int i, j;
const int *p;
p=&j;                                    //允许
p=&i;                                    //允许
i=10;                                    //允许
*P=5;                                    //不允许
```

其中,p 是一个指向常整数的指针,所以修改 p 本身是允许的,但 *P=5 是错误的,而直接对 i 赋值 i=10 却又是允许的。当然,也可以将一个使用 const 修饰的 int 类型的变量的地址赋给 p。例如:

```
const int val=10;
p=&val;
```

这时,不论通过 p 还是直接访问 val 都不能修改 val 的值。

如果指针是指向常量数据的非常量指针,那么不能通过该指针修改它指向的数据。在函数传递参数时,如果形参是指向常量数据的非常量指针,又由于数组名是一个常量指针,指向了数组的第一个元素,那么对应的实参也可以是数组。函数内可以通过形参用数组下标方式访问数组元素,但不能修改每个元素。

3. 指向非常量数据的常量指针

如果想声明一个不可修改的指针变量,保证该指针总是指向固定的内存地址,则需要定义常量指针。

```
int var;
int *const p=&var;
```

这时必须在声明 p 的同时对 p 进行初始化。通过这个声明,p 被定义为一个指向整数的常量指针。这意味着 p 指向的整数通过 p 是可修改的,但是 p 本身的值在初始化后,就不能再修改了,也就是说,p 只能永远指向变量 var。另外,由于声明了 p 是指向整型变量的,也不能将 const 修饰的整型变量的地址赋给 p。

```
const int var;
```

```
int * const p= &var ;                          //错误!
```

因为一旦这样做了,通过指针就可以修改变量 var,这与在对变量 var 进行声明时使用了 const 是矛盾的。

指向非常量数据的常量指针总是指向固定的内存地址,该地址中的数据可以通过指针修改。数组名就是一个常量指针,指向了数组的第一个元素。在函数传递参数时,如果形参是指向非常量数据的常量指针,那么对应的实参可以是数组,函数内可以通过形参用数组下标方式访问并修改数组元素。

4. 指向常量数据的常量指针

可以使用 const 同时限定指针和指针指向的数据,考虑下面的语句:

```
const int val= 10;
const int * const p= &val;
```

这两个语句作声明:val 是一个常整型变量,初始化后不可再修改;p 是一个指向常整数的常量指针,p 本身在初始化指向 val 后也不能被修改,同时也不能通过 p 修改它所指向的变量。

在函数传递参数时,如果形参是指向常量数据的常量指针,那么对应的实参可以是数组,函数内可以通过形参用数组下标方式访问数组元素,但不能修改。

5. const 用于传递函数参数

C++ 有两种传递参数的方法:传值和传引用。两种方式各有优缺点:传值不会破坏实参,但传递参数时要复制数据,如果参数存储空间较大,在时间和空间上效率较低;传引用传递的数据量小,在时间和空间上效率较高,但是被调用函数可以操作实参,可能带来安全性隐患。如果函数参数是指针,可以通过合理使用 const 限定符,既保证实参的安全,又避免因复制大量的数据而带来的时空开销。例如:

```
void output(const double * pd)
{
    cout << * pd;                          //允许
     * pd= 15.5;                           //不允许!
}
```

当然,也可以用 const 关键字来修饰引用参数。例如:

```
void output(const double &d)
{
    cout<<d;                               //允许
    d= 15.5;                               //不允许!
}
```

上面两段程序通过对指针和引用参数使用 const 限定符,一方面是提高参数的传递效率,如双精度类型的变量占用 8 个字节,如果采用传值方式,需要的时空开销较大,而采用传递指针或传引用方式,则只需传递一个地址,一般来说是 4 个字节;另一方面,const 限定符保证了函数内不能对实参进行修改。

【例 7-8】 const 限定符的使用示例。

```
//ex7_8.cpp:  const 限定符的使用
```

```
#include<iostream.h>
#include<string.h>

void printString(const char * );
void toUpper(char * );

int main()
{
    char string[]="this is a demo string.";

    cout<<"The string before conversion is:\n";
    printString(string);
    cout<<endl;

    toUpper(string);
    cout<<"\nThe string after conversion is:\n";
    printString(string);
    cout<<endl;

    return 0;
}
//sPtr 指向常量字符的非常量指针,sPtr 本身可修改,
//但只能访问所指的字符,而不能通过 sPtr 修改它指向的字符
void printString(const char * sPtr)
{
    while (* sPtr!='\0')
    {
        cout << * sPtr;
        sPtr ++;                                //修改 sPtr,指向下一个字符
    }
}
//sPtr 指向非常量字符的非常量指针,不但 sPtr 本身可修改,
//也可以通过 sPtr 访问并修改它所指的字符
void toUpper(char * sPtr)
{
    while (* sPtr!='\0')
    {
        if (* sPtr>='a' && * sPtr <='z')
          * sPtr = * sPtr+ 'A'- 'a';            //转换成大写字符

        sPtr ++;                                //修改 sPtr,指向下一个字符
    }
}
```

程序执行结果：

```
The string before conversion is:
this is a demo string.

The string after conversion is:
```

```
THIS IS A DEMO STRING.
```

7.6 指针和引用

第5章已讨论过传值调用与传引用调用的问题,本节主要介绍指针参数的传递。

指针作为一种变量,可以采取传值和传引用两种方式进行传递。例5-3通过传引用的参数传递方式实现两个整数的交换。下面通过指针,采取传值方式实现该功能。

【例7-9】 用指针采取传值方式实现两个整数的交换。

```
//ex7_9.cpp:用指针采取传值方式实现两个整数的交换
#include<iostream.h>

void swap(int * , int * );

int main()
{
    int x=10, y=20;
    swap(&x, &y);
    cout<<"x:"<<x<<"    y:"<<y<<endl;

    return 0;
}

void swap(int * x, int * y)
{
    int temp;
    temp = * x;
     * x = * y;
     * y=temp;
}
```

程序执行结果:

```
x:20   y:10
```

传递指针也实现了两个整数的交换,但是和例5-3采用传引用方式进行参数传递的实现相比,本例有两处不同:在调用函数时,如使用指针参数,实在参数是变量的地址(通过&运算符),即一个指向整数的指针,而在传引用时实在参数是变量名;在被调用函数内部,如使用指针参数,为了通过形式参数访问主程序中的变量,采取了*运算,而在传引用时直接使用了形式参数名。在本例中,实在参数是变量x和y的地址,这两个地址作为实在参数的值传给了形参a和b,放在了对应的形式单元中。当程序控制转入swap函数后,在执行函数体过程中,对形参的任何引用或赋值都被处理成对相应形式单元的直接访问,所以函数体中为了通过a和b访问到x和y,必须对a和b进行间接访问,即通过*运算访问a和b所指的单元x和y。当被调用函数工作完毕返回时,形式单元所指的x和y中就持有了所期望的值。

可见指针和引用一样,在函数的调用中,被调用函数可以修改调用者的实参,或者通过指针将较大规模的数据对象传递给函数而避免传值调用传递大量数据的开销。

指针和引用是两个不同的概念：指针表示的是对象地址，而引用表示的是对象的别名。指针虽然灵活，但太自由了，它可以指向内存区域的任何位置，很容易造成指针指向系统的保护区域，或对空指针进行间接访问，给程序带来严重后果。因此，指针虽然灵活，却很危险。尽管实际上引用也是借助于指针来实现的，但是引用依附于固定对象，而不能随意指向其他对象，地址的概念对于程序员是透明的，减少了出错的机会。此外，在操纵引用时，语法形式简单，程序也变得易读。

需要指出的是，指针变量本身也可以以传引用的方式进行参数传递，这样在被调用函数中就可以访问并修改调用函数中作为实在参数的指针的值。

7.7　动态内存分配

7.7.1　基本概念

C++ 有两种分配内存的方式：静态分配和动态分配。静态分配是通过变量声明实现的，每声明一个变量，就为这个变量分配了空间。例如，下面的声明：

```
int i, j, k;
float scores[100];
```

都导致了静态内存分配。在程序运行时也可以分配内存空间，这种分配方式称为动态内存分配。动态内存分配由程序根据需要在堆区(自由存储区)为数据分配内存空间。如果程序使用的数据量很大而且可变时，利用动态内存分配可以有效地利用内存空间。例如上面定义的 scores 数组，通过声明数组变量分配了 100 个 float，则程序能够在该数组中存放最多 100 个浮点数，如果实际上只有 10 个浮点数需要存放，那么就浪费了 90 个浮点单元，如果实际上有 101 个需要处理，该数组又不够用了。所以，如果采用静态分配，在声明数组时，必须将数组的大小设定成实际处理时可能需要的最大的数目，这就有可能造成内存空间的浪费。更特殊情况下，如果不能确定数组的规模，又如何声明数组呢？这时，采取动态分配是合适的选择：在运行时，根据实际需要动态分配内存空间。

使用动态内存必须遵循以下 4 个步骤。

① 确定需要多少内存。

② 分配所需的内存。

③ 使用指针指向获得的内存空间。

④ 使用完后，及时释放这部分内存。

动态分配的内存在使用完后要及时释放，否则会使得系统可分配和利用的内存空间不断减少直至枯竭，导致系统无法正常工作。堆区的动态分配和释放有两种方法：一种是利用标准函数，如 malloc 和 free，它们的原型在头文件 stdlib.h 或 alloc.h 中，这是从 C 语言保留下来的方法；另一种是用 new 和 delete 运算符，这是 C++ 的方法，后者比前者功能强，使用更方便。下面介绍这两种方法。

1. 通过 malloc 和 free 动态分配和释放内存

1) malloc 函数动态分配内存

函数 malloc 的原型是：

```
void * malloc(unsigned size);
```

malloc函数分配size个字节的内存空间,并返回指向所分配内存的void *类型的指针。void *指针具有很好的通用性,可以通过类型转换赋值给任何类型的指针变量。如果没有内存空间,返回NULL。例如:

```
int * pn= (int * )malloc(sizeof(int));
```

该语句按照int类型数据存储空间的大小(如果int占4个字节,sizeof(int)的值为4)分配了4个字节的空间,并由整型指针pn指向该区域,将来可以通过pn访问该区域。下面的语句使用了新分配的空间:

```
* pn= 0;
* pn ++ ;
```

通过malloc还可以根据某些变量的值动态分配内存,实现动态数组。例如,如果已知整型变量studNum中保存了全班学生的人数,现在要为全班的某次考试成绩分配一个数组,则可以用下面的代码来实现:

```
int * scores= (int * )malloc(sizeof(int) * studNum);
```

然后,根据数组和指针的关系,可以这样输入每个学生的成绩,并保存到新分配的studNum个整数的空间中:

```
for (int i= 0; i< studNum; i ++ )
cin>> scores [i];
```

与前面定义的数组scores相比,动态分配可以根据运行时的实际需要分配空间,没有空间的浪费,而使用起来又和数组一样方便。

程序设计技巧提示:调用malloc分配内存之后,一定要判断函数返回结果是否为NULL,如果为NULL,则应当进行特殊处理,不能按照原来的程序逻辑继续执行,否则会导致访问内存单元错误。

2) free函数释放内存

函数free的原型是:

```
void free(void * ptr);
```

free函数释放先前malloc所分配的内存,所要释放的内存由指针ptr指向。被释放的内存交还给系统,以便以后重新分配。

程序设计技巧提示:一个指针指向的空间已经通过free释放后,指针本身的值并未改变,但如果试图再次为该指针调用free则会引发错误。所以,将一个指针指向的空间释放后,要主动将该指针置为NULL,以表示该指针未指向任何空间。

2. 通过new和delete动态分配和释放内存

除了保留了C的内存分配函数以外,C++还提供了两个运算符进行动态分配和释放内存:new和delete。它们都是一元运算符,优先级参见附录A。

从上面的例子中可以看到,malloc分配空间的模式是:

```
TypeName * p= (TypeName* ) malloc(sizeof(TypeName));
```

TypeName是任何类型名。malloc需要显式地使用sizeof计算类型占据的字节数，而且返回结果要显式地转换成TypeName*的形式。如果使用new运算符则很简单：

```
TypeName * p= new TypeName;
```

1）new运算符动态分配内存

new运算符自动根据操作数TypeName的字节数分配好空间，并返回正确类型的指针。在后面的章节还可看到，如果TypeName是一个类名，new TypeName还会自动调用该类的构造函数，动态建立一个对象。所以new比malloc具有更强的功能。

同样，new也可以分配动态数组，仍然是上面全班考试成绩的问题，可以这样做：

```
int * scores= new int[studNum];
```

这样分配的内存使用起来和malloc分配的一样：

```
for (int i= 0; i< studNum; i ++)
cin>> scores [i];
```

2）delete运算符释放内存

由new动态分配的内存应该通过delete运算符释放。用delete释放指针p所指的内存的形式为：

```
delete p;
```

如果p是指向通过new动态分配的数组，则采取如下形式：

```
delete [] p;
```

程序设计技巧提示：与free类似，一个指针指向的空间通过delete释放后，指针本身的值也未改变，但若试图再次对该指针使用delete运算，也会引发错误。在用delete释放一个指针后，应主动将该指针置为NULL，表示该指针未指向任何空间。

如果p指向的是一个动态建立的对象，那么delete会自动调用析构函数。

程序设计技巧提示：在C++程序中，动态分配内存既可以通过malloc和free实现，也可以通过new和delete实现。但是最好使用new和delete。

7.7.2　动态分配内存的应用

第6章利用函数对固定元素数目的数组进行了排序，下面分别用malloc和free、new和delete提供的动态内存分配功能，实现对任意数目的整数进行排序。

【例7-10】　用malloc和free动态内存分配方法，实现对任意数目输入整数的排序。

```
//ex7_10: 用 malloc和 free的动态内存分配方法,实现对任意数目的输入整数进行排序
# include< stdlib.h>
# include< iostream.h>

void sortArray(int [], int);

int main()
```

```
{
    int * a;
    int i, num;
    //输入要排序的整数的数目
    cout<<"Please enter the number of integers: ";
    cin>>num;
    //动态分配数组,以保存输入的整数
    a=(int *)malloc(sizeof(int)*num);

    if (a==NULL)
    {
        cout<<"Memory allocating error!Exit."<<endl;
        return 0;
    }
    //输入拟排序的整数
    for (i=0; i<num; i++)
        cin>>a[i];
    //调用函数 sortArray 对 a 进行排序
    sortArray(a, num);
    //输出 a 排序后的结果
    cout<<"After sorting:"<<endl;
    for (i=0; i<num; i++)
        cout<<a[i]<<" ";
    cout<<endl;
    //释放动态分配的空间
    free(a);

    return 0;
}
void sortArray(int b[], int len)
{
    for (int i=0; i<len-1; i++)
    {
        //在未排序序列中找到最小元素
        int min=i;
        for (int j=i+1; j<len; j++)
            if (b[j]<b[min])
                min=j;

        //将找到的最小元素与第 i 个元素交换,即存放到排序序列末尾
        int temp=b[i];
        b[i]=b[min];
        b[min]=temp;
    }
}
```

程序执行结果:

```
Please enter the number of integers: 8
76 45 37 81 10 0 92 128
After sorting:
0 10 37 45 76 81 92 128
```

【例 7-11】 用 new 和 delete 动态内存分配方法，实现对任意数目输入整数的排序。

```
//ex7_11: 用 new 和 delete 的动态内存分配方法,实现对任意数目的输入整数进行排序
#include<stdlib.h>
#include<iostream.h>

void sortArray(int [], int);

int main()
{
    int * a;
    int i, num;

    //输入要排序的整数的数目
    cout<<"Please enter the number of integers: ";
    cin>>num;

    //动态分配数组,以保存输入的整数
    a=new int[num];

    if (a==NULL)
    {
        cout<<"Memory allocating error!Exit."<<endl;
        return 0;
    }

    //输入拟排序的整数
    for (i=0; i<num; i++)
        cin>>a[i];

    //调用函数 sortArray 对 a 进行排序
    sortArray(a, num);

    //输出 a 排序后的结果
    cout<<"After sorting:"<<endl;
    for (i=0; i<num; i++)
        cout<<a[i]<<" ";
    cout<<endl;

    //释放动态分配的空间
    delete [] a;

    return 0;

}

void sortArray(int b[], int len)
{
    for (int i=0; i<len-1; i++)
    {
        //在未排序序列中找到最小元素
```

```
            int min=i;
            for (int j=i+1; j<len; j++)
                if(b[j]<b[min])
                    min=j;
            //将找到的最小元素与第 i个元素交换,即存放到排序序列末尾
            int temp=b[i];
            b[i]=b[min];
            b[min]=temp;
        }
    }
```

程序执行结果:

```
Please enter the number of integers: 8
76 45 37 81 10 0 92 128
After sorting:
0 10 37 45 76 81 92 128
```

从上面的程序可以看出,动态分配的数组在使用方式上与静态数组没有差别,这为程序设计带来了很大的灵活性。动态内存分配是一种非常重要的程序设计技术,在后面的章节中还将看到更多的应用。

*7.8 函数指针

7.8.1 函数指针的定义

虽然函数不是变量,但其代码仍占有存储空间,该存储空间的首地址可以赋给某一指针,这种指针就叫函数指针,它的声明形式如下:

<存储类别> <数据类型> (*<函数指针名>)(<参数表>)

例如:

```
int (*fp)(char);
```

声明 fp 是一个函数指针,它指向的函数返回值为 int 类型,带一个 char 类型的参数。由于函数调用符()的优先级比指针运算符 * 高,因此 *fp 要用括号括起。否则,此声明会变为对指针函数的声明:

```
int *fp(char);
```

声明的是一个函数 fp,该函数返回指向整型数据的指针。

要将函数指针指向某一函数,可以通过赋值语句来实现,假如已定义了函数:

```
int f(char);
```

那么语句

```
fp=f;
```

将使 fp 指向 f，即 fp 中保存了函数 f 的入口地址。如果要将函数指针指向某一函数，应保证函数指针的参数和返回值与要指向的函数相匹配，否则会出错。函数指针与普通变量指针有点不同，前者指向的是程序代码区，而后者指向的是数据存储区。赋值后，可以用对 fp 的调用来代替对函数 f 的调用。

```
int i= (* fp)('J');                    //与语句 i=f('J');等价
```

通过函数指针，可以设计出更加灵活的程序。

7.8.2 函数指针的使用

下面的例子演示了函数指针的使用方法。

【例 7-12】 使用函数指针改进例 7-11，实现按照升序和降序排序。

```
//ex7_12：使用函数指针实现按照升序和降序排序
#include<stdlib.h>
#include<iostream.h>

bool ascending(int, int);
bool descending(int, int);
void sortArray(int [], int, bool (*)(int, int));
void displayArray(int [], int);

int main()
{
    int * a;
    int i, num;
    //输入要排序的整数的数目
    cout<<"Please enter the number of integers: ";
    cin>>num;
    //动态分配数组，以保存输入的整数
    a=new int[num];
    if (a==NULL)
    {
        cout<<"Memory allocating error!Exit."<<endl;
        return 0;
    }
    //输入拟排序的整数
    for (i=0; i<num; i++)
        cin>>a[i];
    //使用 ascending 调用函数 sortArray 对 a 进行排序
    sortArray(a, num, ascending);
    //输出 a 排序后的结果
    cout<<"After ascending sorting:"<<endl;
    for (i=0; i<num; i++)
        cout<<a[i]<<" ";
```

```
    cout<<endl;
    //使用 descending 调用函数 sortArray 对 a 进行排序
    sortArray(a, num, descending);
    //输出 a 排序后的结果
    cout<<"After descending sorting:"<<endl;
    for (i=0; i<num; i++)
        cout<<a[i]<<" ";
    cout<<endl;
    //释放动态分配的空间
    delete [] a;
    return 0;
}
bool ascending(int a, int b)
{
    return (a<b);
}
bool descending(int a, int b)
{
    return (a>b);
}
void sortArray(int b[], int len, bool(* compare)(int, int))
{
    for (int i=0; i<len - 1; i++)
    {
        //在未排序序列中找到排序最靠前的元素
        int head=i;
        for (int j=i+1; j<len; j++)
            if (!(* compare)(b[head] , b[j]))
                head=j;
        //将找到的元素与第 i 个元素交换,即存放到排序序列末尾
        int temp=b[i];
        b[i]=b[head];
        b[head]=temp;
    }
}
```

程序执行结果:

```
Please enter the number of integers: 8
76 45 37 81 10 0 92 128
After ascending sorting:
0 10 37 45 76 81 92 128
After descending sorting:
128 92 81 76 45 37 10 0
```

本例中，将比较大小的功能分别由两个函数 ascending 和 descending 实现，这两个函数的参数和返回值类型完全一致。排序函数 sortArray 接受一个函数指针的参数，主程序可以根据排序的需要，选择 ascending 和 descending 之一来调用 sortArray，以实现不同的排序。

习　题　7

7.1　回答问题：

(1) 什么是指针？指针的值和类型与一般变量有何不同？

(2) 指针具有哪些运算？

(3) 给指针赋值时应注意些什么？使用没有赋过值的指针有什么危险？

(4) 指针作为函数的参数有什么特点？

(5) 一维数组和二维数组的元素如何用指针表示？

(6) 试述引用与指针的异同。

(7) 设有声明：

```
int i=50, * ip=&i;
```

并设变量 i 存放在起始地址为 2500 的存储单元中，则 ip 和 * ip 的值各是多少？

7.2　编写一个程序，按照下列各题的要求，编写一个语句序列，观察程序运行的结果。假设已声明长整型变量 value1 和 value2，value1 初始化为 200000。

(1) 声明变量 lPtr 为 long 类型对象的指针。

(2) 将变量 value1 的地址赋给指针变量 lPtr。

(3) 打印 lPtr 所指的对象值。

(4) 打印 value1 值。

(5) 指定 lPtr 指向变量 value2。

(6) 打印 lPtr 所指的对象值。

(7) 打印 value2 值。

(8) 打印 value2 地址。

(9) 打印 lPtr 中存放的地址，打印的值是否与 value2 的地址相同？

7.3　定义一个整型指针变量 pi，用什么方法，才能使 pi 指向整型变量 i，指向整型一维数组 a 的首地址，指向整型二维数组 b 的首地址？

7.4　下面有两段代码：

(1)

```
int i;
int * p;
i=56;
p=&i;
```

(2)

```
int i;
int * p;
p=&i;
i=56;
```

问执行到每一段的第 4 行以后，* p 的值是否相同?

7.5　下面是一段代码：

```
float pay;
float * ptr;
pay=2313.34;
ptr=&pay;
```

问以下各表达式的值是什么?

(1) pay

(2) * ptr

(3) &pay

7.6　下列程序有什么作用?

```
#include<iostream.h>

int mystery(const char* );

int main()
{
    char string [80];
    cout<<"Enter a strings : ";
    cin>>string;
    cout<<mystery(string)<<endl;
    return 0;
}

int mystery(const char * s)
{
    for (int x=0; * s != '\0'; s++)
        ++x;
    return x;
}
```

7.7　说明一个可存放 10 个字符串的字符串数组，各字符串由用户输入(假设用户输入的字符串长度上限为 32)，数组中每个元素的大小根据用户输入串的实际长度动态地确定。编写一个函数对该数组进行排序。(提示：排序时无须交换两个串，只交换指向它们的指针即可)

7.8　编写 4 个带有两个整型参数并返回整型值的函数，分别输出其两个参数的和、差、积、商。说明一个含有 4 个元素的函数指针数组，各元素分别指向上述 4 个函数。编写程序，根据用户输入的形如：

```
12+34
```

的二元表达式，输出表达式的运算结果。

7.9　利用动态内存分配重新编写习题 6.5 所要求的程序。要求能处理任意人数的学生成绩，人数由用户输入。

7.10　利用动态内存分配重新编写习题 6.10 所要求的程序。要求能处理任意人数任意门课程的成绩，人数与课程数由用户输入(提示：利用指向指针的指针)。

7.11　编写一个函数：

```
char * ReplaceString(char * str , char * s1, char * s2);
```

该函数使用 s2 替换 str 中的 s1，函数返回替换后串的指针，如果 str 中没有串 s1，函数返回 0。

7.12　编写一个函数：

```
void sortLine(char * line[], int n);
```

参数 line 是一个指向串的指针数组，它指向 n 个串，这个函数对这些串进行排序。

注：在比较了两个串的大小之后需要交换两个串时，只需交换 line 中保存的指向这两个串的指针，而不必交换串本身。

试编写一个程序，从键盘上输入一些单词，使用 sortLine 对这些单词进行排序。

7.13　编写一程序，请求用户输入 10 首歌名，歌名存入一字符指针数组，然后分别按原序、字母序和字母逆序(从 Z 到 A)显示这些歌名。

7.14　设计程序，编写加法函数 add()与减法函数 sub()。在主函数中定义函数指针变量，用函数指针变量完成两个操作数的加、减运算。

第 8 章 结构、联合、枚举

【学习内容】

本章介绍 C++ 的结构、联合和枚举。主要内容包括：

◆ 结构的定义与使用。

◆ 联合的定义与使用。

◆ 枚举的定义与使用。

【学习目标】

◆ 能够建立和使用结构、联合、枚举。

◆ 掌握通过传值、传引用、传指针等方式为函数传递结构参数的方法。

高级语言的一个重要特点是允许用户定义自己的数据类型。也就是说，用户可以在高级语言提供的现有类型的基础上，根据实际需要，构造出新的数据类型。用户自定义的数据类型与预定义的数据类型在使用上是一样的，也可以成为构造新的数据类型的基础。自定义数据类型的构造反映了对问题的逐级抽象。C++ 提供了构造用户自定义类型的机制：结构、联合、枚举、类型定义 typedef 和类。

结构、联合和枚举不是 C++ 中所特有的。但是，C++ 对这 3 种语言机制进行了扩充，使其支持面向对象的编程，并在灵活性上有了很大提高。

8.1 结　构

考虑下面的实际问题，一个学生具有学号、姓名、出生日期、性别、考试成绩等数据，这些数据可以用不同的数据类型的数据来表示，如学号可以用整型数据、姓名可以用字符数组或字符串、性别用字符数据、考试成绩可以用浮点数来分别描述，以及出生日期可以用年、月、日 3 个整数构成，但是因为类型不同，所以这些数据不能存放在一个数组中以体现它们在逻辑上的相关性。C++ 为此提供了结构这一机制，专门用于描述类型不同、但逻辑意义相关的数据构成的聚集。所以，结构和数组类似，都是由一组元素构成的聚合数据类型。但是，结构允许这些元素具有不同的数据类型。结构是由用户定义的数据类型，它的成员可以是任何类型，既可以是 C++ 的基本数据类型，如整型、浮点型、字符型和布尔型等，也可以是复杂的数据类型，如数组、指针及其他结构等。C++ 中的结构与类是非常相似的。C++ 中的结构类型可以像类一样处理，这样就可利用 C++ 访问说明符 public、private 和 protected 对

结构成员进行更复杂的访问控制。而这正是C++中结构的独特之处。

8.1.1 结构的定义

在C++中定义结构的一般形式为：

```
struct <结构名>  {<成员列表>};
```

struct是关键字，表示结构定义；<结构名>给出了结构的标识；<成员列表>由若干个成员的声明组成，每个成员都是该结构的一个组成部分，成员也称为字段或域。对每个成员也必须作类型说明。

结构名和成员名由标识符充当。例如：

```
struct student
{
    int num;
    char name[20];
    char sex;
    float score;
};
```

在这个结构定义中，结构名为student，该结构由4个成员组成，包含一个int成员num、一个字符数组成员name、一个char成员sex和一个float成员score。注意，标志结构定义结束的花括号后的分号是不可少的。上述结构的定义，并未在内存中分配任何空间，而是定义了一个新的数据类型。

结构定义之后，即可利用结构名来声明结构类型变量，凡声明为结构student类型的变量都由上述4个成员组成。声明结构类型的变量和声明基本类型的变量的语法一样：

```
student stud1, stud2;
```

定义结构类型的变量时才导致内存的分配。结构类型变量是结构的实例，为其分配空间实际上是为该变量的每个成员分配空间。上面的声明定义了两个student类型的变量stud1和stud2，这两个变量都有自己的4个成员。

从这个例子可以看出，结构是一种复杂的数据类型，是由数目固定、类型不同的若干成员构成的集合。

在student结构定义中，所有的成员都是基本数据类型或数组类型。成员其实也可以是一个结构(但不能是该结构类型自身，即结构的成员不能为该结构自身的实例)，这样就构成了嵌套的结构。图8-1给出了另一个数据结构。

<table>
<tr><td rowspan="2">num</td><td rowspan="2">name</td><td rowspan="2">sex</td><td colspan="3">birthday</td><td rowspan="2">score</td></tr>
<tr><td>year</td><td>month</td><td>day</td></tr>
</table>

图8-1 嵌套结构

按照图8-1可以如下定义结构。

```
struct date
```

```
{
    int year;
    int month;
    int day;
};
struct
{
    int num;
    char name[20];
    char sex;
    date birthday;
    float score;
} stud3, stud4;
```

这里首先定义一个结构 date,由 year(年)、month(月)和 day(日)3 个成员组成。接下来定义了一个结构,这个结构没有名字,该结构中有一个 date 结构类型的成员 birthday,在定义这个结构的同时,声明了两个该结构类型的变量 stud3 和 stud4。这样,变量 stud3 和 stud4 各有 5 个成员,其中成员 birthday 又是结构 date 的实例,它又包含 3 个成员。

成员名可以与程序中其他变量同名,因为 C++ 将它们处理成不同的作用域。

在声明结构变量时可以同时对变量进行初始化,如同对数组变量进行初始化一样,可以在变量名后用等号连接一个用花括号括起来的初始值列表,初始值列表按照结构成员声明的顺序给出了各成员的初始值,各初始值之间用逗号分隔。例如,下面是声明一个 student 结构类型的变量并作初始化:

```
student stud5={ 102, "Li Xiaoming", 'M', 92 };
```

这样声明了一个 student 结构类型的变量 stud5,它的成员 num 初始化成 102,字符数组 name 的内容初始化成"Li Xiaoming",字符成员 sex 初始化成'M',score 的值初始化成 92。如果初始值的数目少于成员数目,剩余的成员被自动初始化成 0,如果成员是指针,则初始化成 NULL。

需要指出,在 C 中,定义一个结构和定义一个结构变量时都必须使用 struct 关键字,如声明 student 结构类型的变量 stud1 和 stud2,必须用语句

```
struct student stud1, stud2;
```

然而在 C++ 中则不用这么麻烦。在 C++ 中,只要在声明结构时使用了 struct 关键字,在其他地方,如定义一个结构变量时只需使用结构名就可以了。如上面例子中成员 birthday 的声明时,就只使用了结构名 date。

但 C++ 编译器仍然能够接受 C 的语法,因而过去的 C 代码可不必修改,这给程序员带来了很大的方便。

8.1.2　结构变量成员的引用

在程序中使用结构变量时,可以把它作为一个整体来使用,如允许具有相同类型的结构变量相互赋值,以上面定义的 stud1 和 stud2 为例,可以通过赋值语句:

```
stud1=stud2;
```

实现将 stud2 的值赋给 stud1，实际是将 stud2 各成员的值赋给 stud1 的对应成员。

更多情况下，需要访问结构变量的各个成员。在 C++ 中，大多数对结构变量的使用，包括输入、输出等都是通过结构变量的成员来实现的，赋值既可以通过上面介绍的相同类型的结构变量相互赋值实现，也可以直接通过对各成员的赋值来实现。访问结构变量的成员时，需要使用成员访问运算符：圆点运算符(.)和箭头运算符(－＞)。

1. 圆点运算符

圆点运算符用于通过结构变量名访问结构成员。结构变量成员的一般表示形式是：

<结构变量名>.<成员名>

例如：

```
stud1.num                    //第一个学生的学号
stud2.sex                    //第二个学生的性别
```

如果成员本身又是一个结构，则可以通过逐级使用圆点运算符找到各级的成员。如 stud3.birthday.month 表示访问前面定义的学生 stud3 的生日的月份成员。结构成员在程序中的使用方式，与普通变量相同。

2. 箭头运算符

箭头运算符由负号(－)和大于号(＞)组成，中间不能有空格。箭头运算符用于通过指针访问指针所指的结构的成员。假设指针 pStud 是一个结构 student 类型的指针，若将结构变量 stud3 的地址赋给 pStud，要打印输出 pStud 指向的结构变量的 num 成员，则可以用下列语句实现：

```
pStud=&stud3;
cout<<pStud->num;
```

表达式 pStud－＞num 等价于(＊pStud).num，后者间接引用指针并用圆点运算符访问 num 成员。这里必须有括号，因为圆点运算符的优先级高于指针的间接引用运算符＊。

8.2 结构的使用

8.2.1 结构与函数

结构类型数据(包括结构变量和结构数组的元素)的成员在使用方式上与同类型的一般变量一样，也可以作函数的参数和返回值。而结构型数据本身也可以作为函数的参数和返回值来使用。

1. 结构型数据的成员作函数的参数

如果函数的形参是简单变量，结构型数据中相应类型的成员可作为对应的实参。参数传递的方式与一般变量相同。如果函数的形参是指针类型或数组类型，结构型数据中同类型的成员也都可作为对应的实参。例如，考虑前面定义的结构 student 及其变量 stud：

```
struct student
{
    int num;
    char name[20];
    char sex;
    float score;
} stud;
```

假设有若干函数,其原型如下:

```
void f1(int);
void f2(float);
void f3(char * );
```

那么下面的函数调用是合法的。

```
f1(stud.num);                 //传递 stud.num 的值
f2(stud.score);               //传递 stud.score 的值,即赋值调用
f3(stud.name);                //传递数组 stud.name 的值,即第一个元素的地址
f1(stud.name[2]);             //传递数组元素 stud.name[2]的值
```

如果要将结构型数据的成员的地址传送给函数,就必须在表示结构成员的表达式前使用取地址运算符 &;同时,函数的形参也应该是指针或数组类型。例如,假设另有两个函数原型如下:

```
void f4(int * );
void f5(float * );
```

那么,下面的调用形式是正确的:

```
f3(stud.name);                //传递数组 stud.name 的值,即第一个元素的地址
f3(&stud.name[2]);            //传递数组元素 stud.name[2]的地址
f4(&stud.num);                //传递成员 stud.num 的地址
f5(&stud.score);              //传递成员 stud.score 的地址
```

注意:运算符 & 放在结构型数据名前,而不是放在成员名之前。数组名 name 本身就是数组的首地址,相应的结构名前不需要用 &,但取数组元素 name[2]的地址时,结构名前要使用 &。

2. 结构型数据作函数的参数或返回值

如果结构变量作为实参传给函数,将按照传值的方式将整个结构型数据复制给相应的形参,传送给被调用的函数。采取这种方式,使得被调用函数不能直接通过形参访问实参的结构型数据。一般情况下,由于结构变量的成员较多,占用内存也较多,传送时要花费很多时间,所以不提倡这种方法。例如,假设有函数:

```
void f6(student s);
```

调用形式为:

```
f6(stud);
```

传递参数时，是将实参 stud 每个成员的数据复制给形参 s 相应的成员，因此要消耗一定的时间和空间。在函数调用时如果要传递结构类型的参数，常用的方法是采取传引用的方式或者传递指针参数。采取这两种方式，不需要对实参的成员进行复制，仅传递结构的地址，占用内存少，调用速度快。例如，下面的函数调用具有较高的效率。

函数原型为：

```
void f7(student &s);
void f8(student * s);
```

相应的调用形式为：

```
f7(stud);
f8(&stud);
```

结构也可作为函数的返回值。下面的函数原型声明的函数具有不同的返回类型：

```
student * func1(void);                    //返回结构指针
student func2(void);                      //返回结构类型变量
```

【例 8-1】 结构作为函数的参数示例。

```
//ex8_1.cpp: 结构作为函数的参数示例
#include<iostream.h>

struct student
{
    int num;
    char name[20];
    char sex;
    float score;
};

void funcCallByValue(student);
void funcCallByReference(student &);
void funcCallByPointer(student * );
void displayStudentInfo(const student &);

int main()
{
    student theStud={ 102, "Li Xiaoming", 'M', 92 };

    cout<< "Initial information:";
    displayStudentInfo(theStud);

    funcCallByValue(theStud);
    cout<< "\nAfter call by value:";
    displayStudentInfo(theStud);

    funcCallByReference(theStud);
    cout<< "\nAfter call by reference:";
    displayStudentInfo(theStud);
```

```
    funcCallByPointer(& theStud);
    cout<<"\nAfter call by pointer:";
    displayStudentInfo(theStud);
    return 0;
}
void displayStudentInfo(const student & stud)
{
    cout<<endl;
    cout<<"num="<<stud.num<<"\t";
    cout<<"name="<<stud.name<<"\t";
    cout<<"sex="<<stud.sex<<"\t";
    cout<<"score="<<stud.score<<endl;
}
void funcCallByValue (student stud)
{
    stud.score++;
}
void funcCallByReference (student &stud)
{
    stud.score++;
}
void funcCallByPointer (student * stud)
{
    stud->score++;
}
```

程序执行结果:

```
Initial information:
num=102         name=Li Xiaoming        sex=M score=92
After call by value:
num=102         name=Li Xiaoming        sex=M score=92
After call by reference:
num=102         name=Li Xiaoming        sex=M score=93
After call by pointer:
num=102         name=Li Xiaoming        sex=M score=94
```

从上面的程序运行结果可看出,采取传值方式时,对形参的成员的访问不会影响实参的结构,而传引用和传指针,被调用函数都可以访问并修改实参结构的成员。当结构包含大量的数据时,传值方式将影响程序的运行效率,这时应该采用传引用或传指针参数的方式。如程序中 displayStudentInfo 函数就是采取传引用的方式传递结构。同时,为防止传引用时修改实参的值,该函数的形参使用了 const 关键字,以确保函数内部除访问实参结构的成员外,不会修改它们的值。另外,函数 funcCallByReference 也采取传引用的

方式，而函数 funcCallByPointer 则采取指针参数。注意这两个函数的区别：采取传引用方式，实参是结构变量本身，被调用函数内部访问引用变量所引用的实参的数据成员时，使用成员运算符.；而使用指针参数时，实参是结构变量的地址，被调用函数内部访问实参结构的成员时，使用了运算符－＞。相对于传值方式，传引用方式或传指针参数，速度更快，程序效率更高。

【例 8-2】 函数返回结构的使用示例：编写一个函数，从键盘接受用户输入的学生的学号、姓名、性别和成绩等信息，将这些信息保存在结构中并返回给主程序。要求主程序完成结构的初始化，接收函数返回的结构并输出新输入的学生的信息。

```
//ex8_2.cpp:函数返回结构的使用示例
# include< iostream.h>

struct student
{
    int num;
    char name[20];
    char sex;
    float score;
};

student getStudent();
void displayStudentInfo(const student & stud);

int main()
{
    student theStud= { 102, "Li Xiaoming", 'M', 92 };

    cout<< "Initial student information:";
    displayStudentInfo(theStud);

    theStud= getStudent();
    cout<< "\nAfter call getStudent:";
    displayStudentInfo(theStud);

    return 0;
}

student getStudent()
{
    student stud;

    cout<< "Please enter the number: ";
    cin>> stud.num;
    cout<< "Please enter the name: ";
    cin>> stud.name;
    cout<< "Please enter the sex: ";
    cin>> stud.sex;
    cout<< "Please enter the score: ";
    cin>> stud.score;
    return stud;
```

```
}
void displayStudentInfo(const student & stud)
{
    cout<<endl;
    cout<<"num="<<stud.num<<"\t";
    cout<<"name="<<stud.name<<"\t";
    cout<<"sex="<<stud.sex<<"\t";
    cout<<"score="<<stud.score<<endl;
}
```

程序执行结果：

```
Initial student information:
num=102        name=Li Xiaoming       sex=M score=92
Please enter the number: 105
Please enter the name: John
Please enter the sex: M
Please enter the score: 95
After call getStudent:
num=105        name=John      sex=M score=95
```

函数 getStudent 将一个结构返回给主函数 main，函数 main 接收到该结构后，通过结构的赋值运算，将返回的结构的各个成员值赋给结构 theStud 的对应成员。因此，从最后输出的结果可以看到结构 theStud 的成员的值发生了变化。

8.2.2 结构与数组

数组的元素也可以是结构类型的，可以设计结构数组。结构数组的每一个元素都是相同类型的结构变量，即每个下标变量都代表一个结构变量。在实际应用中，经常用结构数组来表示具有相同数据结构的一个群体。例如，班级的学生档案、公司职工的工资表、银行的客户信息等。结构数组的处理方法与一般数组类似，结构数组的元素的处理也与一般的结构变量相似。例如：

```
struct student
{
    int num;
    char name[20];
    char sex;
    float score;
}class[5];
```

定义了一个结构数组 class，共有 5 个元素，class[0]～class[4]，每个数组元素都是 student 类型的结构变量。结构数组在声明时可以进行初始化。例如：

```
struct student
{
    int num;
```

```
    char name[20];
    char sex;
    float score;
} class[5]={
    {110, "Zhang Ping", 'M', 45},
    {102, "Li Xiaoming", 'M', 92},
    {153, "Wang Ming", 'M', 52.5},
    {134, "Cheng Ling", 'F', 87},
    {105, "Wang Xiaofang", 'F', 95},
};
```

上面的声明将 class 数组的 5 个元素进行了初始化。如果不给出数组长度，那么初始化值的数目决定了数组元素的数目。

【例 8-3】 编写一个程序，将学生的档案记录按照学号从小到大进行排序。

设计思想：本例用结构数组来保存学生的档案信息，然后用函数 displayStudentsInfo 来显示学生的信息，用函数 sortArray 实现档案记录按照学号从小到大排序，排序使用的是直接选择排序方法。

```
//ex8_3.cpp: 将学生的档案记录按照学号从小到大进行排序
#include<iostream.h>
#define STUDENT_Num 5
struct student
{
    int num;
    char name[20];
    char sex;
    float score;
};

void displayStudentsInfo(const student [], int);
void sortArray(student [], int);

int main()
{
    //初始化
    student theClass[STUDENT_Num]={
                        {110, "Zhang Ping", 'M', 45},
                        {102, "Li Xiaoming", 'M', 92},
                        {153, "Wang Gang", 'M', 52.5},
                        {134, "Cheng Ling", 'F', 87},
                        {105, "Liu Xiaofang", 'F', 95},
                    };
    cout<< "Initial student information:\n";
    displayStudentsInfo(theClass, STUDENT_Num);

    cout<< "\nAfter sorting:\n";
    sortArray(theClass, STUDENT_Num);
```

```
        displayStudentsInfo(theClass, STUDENT_Num);
        return 0;
}
void displayStudentsInfo(const student studs[], int len)
{
        for (int i=0; i<len; i++)                        //扫描一遍
        {
                cout<<"num="<<studs[i].num<<"\t";
                cout<<"name="<<studs[i].name<<"\t";
                cout<<"sex="<<studs[i].sex<<"\t";
                cout<<"score="<<studs[i].score<<endl;
        }
}
void sortArray(student studs[], int len)
{
        for (int i=0; i<len-1; i++)
        {
                //在未排序序列中找到学号最小的元素
                int min=i;
                for (int j=i+1; j<len; j++)
                        if (studs[j].num<studs[min].num)
                                min=j;
                //将找到的学号最小的元素与第 i 个元素交换,即存放到排序序列末尾
                student temp=studs[i];
                studs[i]=studs[min];
                studs[min]=temp;
        }
}
```

程序执行结果:

```
Initial student information:
num=110     name=Zhang Ping      sex=M score=45
num=102     name=Li Xiaoming     sex=M score=92
num=153     name=Wang Gang       sex=M score=52.5
num=134     name=Cheng Ling      sex=F score=87
num=105     name=Liu Xiaofang    sex=F score=95
After sorting:
num=102     name=Li Xiaoming     sex=M score=92
num=105     name=Liu Xiaofang    sex=F score=95
num=110     name=Zhang Ping      sex=M score=45
num=134     name=Cheng Ling      sex=F score=87
num=153     name=Wang Gang       sex=M score=52.5
```

函数 displayStudentsInfo 和 sortArray 都接收结构数组,其中前者只是访问而不修改实参结构数组,而后者修改实参。这两个函数内部在使用结构数组及其元素、访问结构的成员等方面,并没有特殊的地方。

8.2.3 结构与指针

1. 结构指针变量的声明

当一个指针变量指向一个结构时，称之为结构指针变量，或结构指针。结构指针变量的值是它所指向的结构的首地址。通过结构指针即可访问它所指向的结构及其成员，这与数组指针和函数指针的情况是类似的。结构指针变量声明的一般形式为：

```
<结构名> *<结构指针变量名>;
```

＜结构名＞为指针指向的结构的名字。例如：

```
student *pStud, stud;
pStud=&stud;
```

声明了一个结构变量 stud 和一个结构指针 pStud，该指针指向了结构 stud。

在 C 语言中声明结构指针变量时，在＜结构名＞前面还要有关键字 struct：

```
struct <结构名> *<结构指针变量名>;
```

与前面介绍的各种指针变量相同，结构指针变量也必须初始化后才能使用。可以把结构变量的首地址赋予该指针变量，注意不能把结构名赋予该指针变量。因为结构名和结构变量是两个不同的概念，不能混淆。结构名只表示一个结构形式，编译系统并不对它分配内存空间，将其视为一种数据类型。只有当某变量被声明为这种类型的结构时，才对该变量分配存储空间。

2. 结构指针变量的使用

有了结构指针变量，就可以通过指针方便地访问它所指向的结构变量的各个成员。访问的一般形式为：

```
(*<结构指针变量>).<成员名>
```

或者

```
<结构指针变量>-><成员名>
```

例如：

```
(*pStud).num
```

或者

```
pStud->num
```

应该注意(*pStud)两侧的括号不可少，因为圆点运算符.的优先级高于*。如果去掉括号写作*pStud.num，则相当于*(pStud.num)，意义就完全不同了。

【例 8-4】 结构指针变量的声明和使用示例。

```
//ex8_4.cpp:结构指针变量的声明和使用示例
#include<iostream.h>

struct student
{
    int num;
    char name[20];
    char sex;
    float score;
};

int main()
{
    //初始化
    student stud={102, "Li Xiaoming", 'M', 92};
    student *pStud=&stud;

    //通过结构变量访问结构的成员
    cout<<"Access structure through structure variable:\n";
    cout<<"num="<<stud.num<<"\t";
    cout<<"name="<<stud.name<<"\t";
    cout<<"sex="<<stud.sex<<"\t";
    cout<<"score="<<stud.score<<endl;

    //通过指针访问结构,使用圆点运算符结构的成员
    cout<<"Access structure through pointer and (.):\n";
    cout<<"num="<<(*pStud).num<<"\t";
    cout<<"name="<<(*pStud).name<<"\t";
    cout<<"sex="<<(*pStud).sex<<"\t";
    cout<<"score="<<(*pStud).score<<endl;

    //通过指针访问结构,使用箭头运算符访问结构的成员
    cout<<"Access structure through pointer and (->):\n";
    cout<<"num="<<pStud->num<<"\t";
    cout<<"name="<<pStud->name<<"\t";
    cout<<"sex="<<pStud->sex<<"\t";
    cout<<"score="<<pStud->score<<endl;

    return 0;
}
```

程序执行结果:

```
Access structure through structure variable:
num=102        name=Li Xiaoming        sex=M score=92
Access structure through pointer and (.):
num=102        name=Li Xiaoming        sex=M score=92
Access structure through pointer and (->):
num=102        name=Li Xiaoming        sex=M score=92
```

本例中,程序定义了一个结构 student,并声明了 student 结构类型的变量 stud 和指向

student 结构的指针 pStud,pStud 指向了 stud。程序中使用 3 种形式输出 stud 的各个成员的值。从运行结果可以看出,访问结构成员的方式有 3 种:

```
<结构变量>.<成员名>
(*<结构指针变量>).<成员名>
<结构指针变量>-><成员名>
```

它们是完全等效的。

3. 结构数组与指向结构的指针

在第 7 章中看到,C++ 中指针和数组的联系非常紧密,有时二者可以互换使用。结构数组与指向结构的指针也是如此。结构类型的指针可指向一个结构数组,这时结构指针的值是整个结构数组的首地址。结构类型的指针也可指向结构数组中任意一个元素,这时结构指针变量的值是该结构数组元素的首地址。假设 ps 为指向结构数组的指针变量,则 ps 也指向该结构数组的 0 号元素,ps+1 指向 1 号元素,ps+i 则指向 i 号元素。这与普通数组的情况一致。下面的例子说明了结构数组与指向结构的指针在使用上的灵活性。

【例 8-5】 通过指针变量访问并输出结构数组的各元素。

```
//ex8_5.cpp:通过指针变量访问并输出结构数组的各元素
#include<iostream.h>
#define STUDENT_Num 5
struct student
{
    int num;
    char name[20];
    char sex;
    float score;
};
void displayStudentInfo(const student * const);
int main()
{
    student theClass[STUDENT_Num]={
                        {110, "Zhang Ping", 'M', 45},
                        {102, "Li Xiaoming", 'M', 92},
                        {153, "Wang Gang", 'M', 52.5},
                        {134, "Cheng Ling", 'F', 87},
                        {105, "Liu Xiaofang", 'F', 95},
                    };
    student * pStud=theClass;
    for (int i=0; i<STUDENT_Num; i++)
        displayStudentInfo(pStud++);
    return 0;
}
```

```
void displayStudentInfo(const student * const stud)
{
    cout<<"num="<<stud->num<<"\t";
    cout<<"name="<<stud->name<<"\t";
    cout<<"sex="<<stud->sex<<"\t";
    cout<<"score="<<stud->score<<endl;
}
```

程序执行结果:

```
num=110        name=Zhang Ping        sex=M score=45
num=102        name=Li Xiaoming       sex=M score=92
num=153        name=Wang Gang         sex=M score=52.5
num=134        name=Cheng Ling        sex=F score=87
num=105        name=Liu Xiaofang      sex=F score=95
```

程序声明了 student 结构类型的数组 theClass,并进行了初始化。在 main 函数内声明 pStud 为指向 student 结构的指针,其值被初始化成数组 theClass 的首地址。在 for 循环语句中,通过 pStud 自增运算,使得 pStud 指向 theClass[i],然后调用函数 displayStudentInfo 输出 theClass 数组中各元素的成员值。

注意,一个结构指针虽然可以用来访问结构变量或结构数组元素的成员,但是不能试图让它指向结构成员,不允许取结构成员的地址来赋予它。下面的赋值是错误的。

```
pStud=&theClass[1].sex;
```

而只能是

```
pStud=theClass;                    //赋予数组首地址
```

或者是

```
pStud=&theClass[0];                //赋予 0 号元素首地址
```

*8.2.4 位段

C++ 是一种便于操作机器的语言,它不仅可以按字节访问基本类型对象,还可以通过位运算处理字节中的每一个二进制位。在结构(以及后面介绍的联合)中,C++ 允许对构成结构(或联合)的成员指定其存储二进制位数,把这种指定了存储位数的成员称为位段(bit field)。利用位段能够根据成员的实际取值范围使用最少的二进制位数来存储数据,从而更好地利用内存。含位段的结构定义的一般形式为:

```
Struct <结构名>{
    <类型名><位段名>:<位段的二进制位数>;
    <类型名><位段名>:<位段的二进制位数>;
    …
    <类型名><位段名>:<位段的二进制位数>;
};
```

位段的类型仅限于各种整型和后面介绍的枚举类型，长度为1的位段必须说明为unsigned，因为位不能只带符号(某些编译器可能仅允许无符号位段)。位段的二进制位数是一个常量表达式，必须大于等于0。通常位数为0的字段都是无名字段，表示结构的下一个字段从新的存储单元的起始位置开始。例如：

```
struct student
{
    int num: 12;
    unsigned: 0;
    unsigned score: 7;
};
```

宽度为0的无名字段使得其前面的num字段剩下的存储位被跳过，score字段从下一个新的存储单元的起始位置开始存储。

位段的处理与机器有关。有的机器允许位段跨越字的边界，有的却不允许。使用时要参看具体机器的说明资料。

【例8-6】 用位段来存放学生的信息。

```
//ex8_6.cpp: 用位段来存放学生的信息
#include <iostream.h>
#define STUDENT_Num     5
struct student
{
    unsigned short num : 8;
    unsigned short sex : 1;
    unsigned short score : 7;
};
void displayStudentInfo(const student * const);
int main()
{
    student theClass[STUDENT_Num]={
                                {110, 1, 45},
                                {102, 1, 92},
                                {153, 1, 53},
                                {134, 0, 87},
                                {105, 0, 95},
                            };
    student * pStud=theClass;
    cout<<"sizeof(student)="<<sizeof(student)<<endl;
    for (int i=0; i<STUDENT_Num; i++)
        displayStudentInfo(pStud++);
    return 0;
}
void displayStudentInfo(const student * const stud)
```

```
{
    cout<<"num="<<stud->num<<"\t";
    cout<<"sex="<<stud->sex<<"\t";
    cout<<"score="<<stud->score<<endl;
}
```

程序执行结果：

```
sizeof(student)=2
num=110         sex=1 score=45
num=102         sex=1 score=92
num=153         sex=1 score=53
num=134         sex=0 score=87
num=105         sex=0 score=95
```

程序中 student 结构有 3 个 unsigned short 字段 num、sex 和 score，根据这些数据的取值范围，将它们的宽度分别设定为 8、1 和 7。从程序执行效果看，位段在使用上与一般字段是同样方便的。但是，程序执行结果也显示，student 仅占用了 2 个字节，可见位段更能有效地利用存储空间。

位段在某些方面是有限制的，如不能使用位段的地址。此外，位段是按从左到右还是从右到左分配依赖于具体的机器。

也可以将一般的结构成员与位段成员混用。例如：

```
struct employee
{
    float pay;
    unsigned short lay_off:1;                //雇员状态
    unsigned short hourly: 1;                //雇员是否有薪金
    unsigned short deductions: 3;            //扣除的薪金数
};
```

上述结构用一个 float 保存了应付的薪金，而用一个 unsigned short 中的 5 个二进制位保存了后 3 个信息：雇员状态、雇员是否有薪金、扣除的薪金数。与一般结构相比，位段节省了存储空间。

8.3 联　　合

联合是 C 语言特有的一种类型，C++ 也保留了这一机制。联合与结构有许多相似之处：都是用户自定义的类型，都由多个成员组成，成员的类型可以不同。但联合与结构的意义和作用是非常不同的：结构是若干个逻辑上相关的成员的聚集，每个成员都有自己的存储空间；联合不是各个成员的聚集，联合中的各个成员共享存储空间。利用联合，可以将程序当中使用范围不同的变量放在一个联合中，让它们共享存储单元，提高内存的利用效率。一个联合当中常有多个成员，这些成员的类型可以相同也可以不同。

8.3.1 联合和联合变量定义

1. 联合类型的定义

在C++中，不同类型的数据可以共享内存，这可以通过构造联合类型来实现。联合的定义与结构相似，其不同之处仅在于使用内存的方式。联合的一般定义形式如下：

```
union <联合名>  {<成员列表>};
```

union是关键字，表示联合定义。与结构类似，<联合名>给出了联合的标识，<成员列表>由若干个成员的声明组成，每个成员都是该联合的组成部分，也称为字段或域。对每个成员也必须作类型说明，其形式为：

```
<类型说明符><成员名>;
```

联合名和成员名由标识符充当。例如：

```
union number {
    int i;
    char ch;
    float f;
};
```

定义了联合number，联合中的成员i、ch和f共享内存单元。假定i占4个字节，ch占一个字节，f占4个字节，那么3个成员共享内存的方式如图8-2所示。

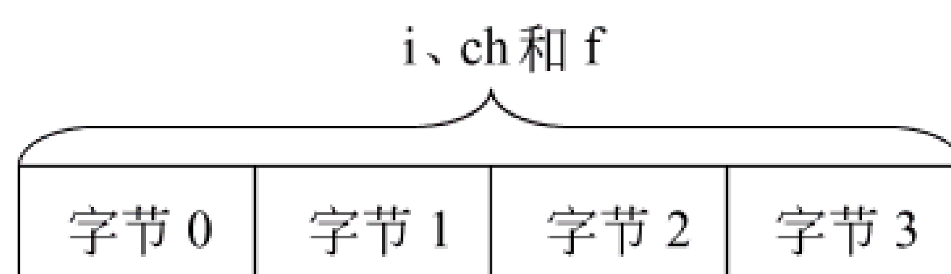


图8-2　联合的成员共享内存示意图

2. 联合变量定义

与结构一样，联合的声明仅定义了一个数据类型，并未分配空间。只有声明联合的变量时才导致内存的分配。联合变量的定义和结构一样，既可以先定义联合类型，然后用该类型名声明变量，也可以联合与变量同时说明。例如：

```
number num;
```

与

```
union number {
    int i;
    char ch;
    float f;
} num;
```

都是允许的。

在定义联合变量时，编译器总是根据联合中占用内存单元最多的那个成员的要求来分配内存。例如，num就按int和float的大小分配空间，即分配4个字节。

联合声明时只能用与第一个成员类型相同的值进行初始化。例如：

```
number num={100};
```

而声明

```
number num={100.5};
```

是错误的。

访问联合中的元素时,也要使用.或—>运算符,其用法与结构类似。例如:

```
num.i=100;
```

访问联合num中的成员i,并在该联合的存储单元中存入整数100的二进制表示。

8.3.2 联合的使用

联合与结构相比有其特殊性,它的所有成员在空间上相互覆盖。因此,可以这样理解,在特定时刻,联合只有一个特定的成员被保存,而不能保存其他成员的信息。联合常用在多种类型的对象不需要同时保存和访问的情况下。例如,编译程序中符号表的数据可以是int型、float型或字符型,但是对于一个符号只能是其中的一种,因此可用联合来存放它们。

```
struct symbol{
    char * name;
    int type;
    union {
        int i;                          //处理整数
        char ch;                        //处理字符数
        float f;                        //处理浮点数
    } value;
}
```

联合value将整数、字符数据和浮点数都放在同一内存区域存储,节省了存储空间。从上面的声明也可以看出联合可以嵌入结构中,即结构的成员类型可以是联合。

与无名结构类似,联合也可以没有名字,这种联合称为无名联合。例如,上面所定义的联合。无名联合只能在定义处说明对象。

【例8-7】 联合类型的定义和使用示例。

```
//ex8_7.cpp:联合类型的定义和使用示例
#include<stdio.h>

struct symbol {
    char * name;
    int type;                       //整数表示的类型信息
    union {
        int i;                      //处理整数
        char ch;                    //处理字符数
        float f;                    //处理浮点数
    } value;
};

int main()
```

```
{
    symbol sym={ "number", 101,  { 122 } };

    printf("The symbol %s is:\n", sym.name);
    printf("type=%d\n", sym.type);

    printf("value in hex is: %x\t", sym.value.i);
    printf("size of sym.value is: %d\n", sizeof(sym.value));

    printf("value.i=%d\n", sym.value.i);
    printf("value.ch='%c'\n", sym.value.ch);
    printf("value.f=%e\n", sym.value.f);

    return 0;
}
```

程序执行结果：

```
The symbol number is:
type=101
value in hex is: 7a      size of sym.value is: 4
value.i=122
value.ch='z'
value.f=1.709584e-043
```

可以看出，结构 sym 的 value 成员占用 4 个字节，value 成员首先被初始化成整数 122，通过 value.i 访问该存储空间得到整数 122，通过 value.ch 访问该存储空间得到字符'z'，而通过 value.f 访问该存储空间得到浮点数 1.709584e－043。注意，整数 122、字符'z'和浮点数 1.709584e－043 的二进制表示都是相同的。

8.4 枚　　举

到目前为止，对于离散的、非数值数据的描述还没有找到令人满意的方法。例如，如果实际问题要表示处理诸如星期、月份之类的数据时，可以在程序中用 0、1、…、6 分别表示星期日、星期一、……、星期六，也可以用 1、2、…、12 表示一月、二月、……、十二月。也就是说，可以利用整型数据来描述这类离散的、非数值数据。那么，既可以用 0、1、…、6 分别表示星期日、星期一、……、星期六，也可以用 10、11、…、16 来表示，因此表示法不是唯一的。在这种表示法下，整型的其他值(如－1、－3 等)又具有什么意义呢？另外，用整数来表示星期、月份之类，为理解程序带来困难，这些代表非数值数据的整数与其他整数在程序中让人难以区分。还有，整数有自己的运算，如加、减、乘、除等，用整数代表非数值数据，如何理解这些运算呢？

用整数来描述非数值数据，其实是在实际数据和整型数据之间建立了映射，而这种映射降低了程序的可读性。同时，也无法确保整数的表示、语义、取值范围、运算以及其他特性可以简单地搬过来。这种办法的局限性是显而易见的。

为了克服上述种种弊端，C++ 语言允许程序员定义新的数据类型，在程序中可以用易于理解的自然语言(如英语)词汇来表示非数值数据。例如，星期一用 Monday 表示，一月用

January表示,……。注意,这里出现的英文字母并非是字符串,也不是某一数据对象的名字,仅仅表示用户自定义的非数值数据。这种可将其非数值数据一一列出的数据类型,称为枚举类型。

8.4.1 枚举和枚举型变量的定义

1. 枚举类型的定义

枚举是用标识符表示的整型常数集合,这些常数是该类型变量可取的合法值。这些标识符称为枚举常量。

枚举的定义是由enum关键字和一个以逗号为分隔符、用花括号括起来的标识符表构成。其声明的一般形式是:

```
enum <枚举名> { <标识符 1> , <标识符 2> ,…,<标识符 n> };
```

花括号中是标识符列表,分别代表对应的整数值。例如:

```
enum weekday { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
```

定义了枚举类型weekday,Sun、Mon、Tue、Wed、Thu、Fri和Sat都表示整常数。在默认情况下,整常数从0开始,因此,Sun=0、Mon=1、Tue=2、Wed=3、Thu=4、Fri=5和Sat=6。也可以根据需要改变枚举常量对应整常数。例如:

```
enum weekday {Sun=1, Mon, Tue, Wed, Thu, Fri, Sat};
```

因为第一个枚举常量被定义为1,所以后续的常量的值依次递增1,分别是2、3、4、5、6、7。实际上,C++允许明确定义每个枚举常量对应的整数值,甚至不同的枚举常量可以取相同的值,又如:

```
enum weekday {Sun=1, Mon, Tue=5, Wed, Thu, Fri=10, Sat};
```

在语法上也是允许的。

枚举可以用来定义各种离散的、非数值型的数据。下面是一些枚举定义的例子:

```
enum  months {January, February, March, April, May, June, July, August, September,
October, November, December};
enum  season {spring, summer, autumn, winter};
enum  colour {red,yellow,blue,white,black};
enum  number {one, two, three, four, five, six, seven, eight, nine, ten);
enum  sex {male, female};
enum  operator {plus, minus, multiply, divice};
```

2. 枚举型变量的定义

有了枚举类型后,就可以定义它的变量。例如,可以使用上面定义的枚举类型weekday声明变量day:

```
weekday day;
```

day只能在weekday的枚举常量范围内取值。如果超出该范围,则会导致编译错误。

8.4.2 枚举类型变量的赋值和使用

枚举类型定义的枚举常量可以作为整数使用。但是，把整数值赋给枚举类型变量必须进行强制类型转换，否则会出错。例如，程序段：

```
enum weekday { Sun, Mon, Tue, Wed, Thu, Fri, Sat } a,b,c;
a=Sun;
b=Mon;
c=Tue;
printf("%d, %d, %d",a,b,c);
return 0;
```

输出的结果是：

```
0, 1, 2
```

注意，不能直接把枚举常量对应的整数值赋给枚举变量。例如：

```
a=Sum;
b=Mon;
```

是正确的。而

```
a=0;
b=1;
```

是错误的。如果一定要把数值赋予枚举变量，则必须用强制类型转换。例如：

```
a=(weekday)2;
```

其意义是将对应的值为 2 的枚举常量赋予枚举变量 a，相当于：

```
a=Tue;
```

说明：枚举常量不是字符常量也不是字符串常量，使用时不要加单、双引号。

如果需要，也可随机指定标识符表中各枚举常量对应的整数值。例如：

```
enum color {red,white=7, black=2, blue};
```

编译器为表中各枚举常量分配的整数值分别为 0、7、2、3。

枚举名也不是必需的，没有名字的枚举类型称为匿名枚举。匿名枚举常用来表示一组常量。例如：

```
enum {MALE, FEMALE};
```

定义了一组整常量，它与下面的定义是等价的。

```
const int MALE=0;
const int FEMALE=1;
```

实践证明，合理地使用枚举类型将会大大地改善程序的可读性。

习 题 8

8.1 回答问题:

(1) 什么是结构? 它和数组有何区别?

(2) 如何对结构变量进行初始化? 如何给结构变量赋值?

(3) 什么是联合? 联合和结构有什么区别?

(4) 枚举类型如何定义? 使用时要注意些什么?

8.2 给出下列结构和联合的定义。

(1) 结构 inventory,包含字符数组 partName[10]、整数变量 parrtNumber、浮点数变量 price、整数变量 stock 和整数变量 recorder。

(2) 联合 data,包含 char c、short s、long l 和 double d。

(3) 结构 address,包含字符数组 streetAddress[32]、city[24]、province[8]和 zipCode[7]。

(4) 结构 student,包含字符数组 firstName[15]和 lastName[15]以及类型(3)中的 struct address 的变量 homeAddress。

(5) 结构 test,包含 16 个宽度为 1 位的位段,位段名从 a 到 p。

8.3 说明一个表示日期的结构 date,其成员包括年、月、日。考虑应为该结构定义哪些操作函数。

8.4 编写一个函数 day,该函数使用 8.3 中的 date 结构作为参数,函数返回某日是这年的第几天。注意闰年问题。

8.5 建立一个联合 union integer,其成员包括 char c、short s、int i 和 long l。编写一个程序,读取 char、short、int 和 long l 类型的值,把它们存入类型为 union integer 的变量中。用对应的类型(char、short、int 和 long 类型)分别打印出联合体的每一个变量。打印出的值都正确吗?

8.6 使用结构变量来表示每个学生的数据:姓名、学号和 3 门课程的成绩。从键盘上输入 10 个学生的数据,要求打印出每个学生的姓名和 3 门课程的平均成绩。

8.7 说明一个包含有 30 个元素的 address_book 型结构数组,考虑应为该数组提供哪些操作函数。试画出一个通讯录结构 address_book 型变量在内存中的存储形式。

8.8 设计一个记录学生学号、姓名和成绩的结构,并通过结构数组将全组 10 名同学的信息输入,完成根据学生姓名查询的程序。

8.9 定义描述复数类型的结构体变量,编写减法函数 sub()与乘法函数 mul(),分别完成复数的减法与乘法运算。在主函数中定义四个复数类型变量 c1、c2、c3、c4,输入 c1、c2 的复数值,调用 sub()完成 c3=c1-c2 操作,调用 mul()完成 c4=c1 * c2 操作。最后输出 c3、c4 复数值。

8.10 定义两个操作结构 worker 的函数 input 和 display。结构 worker 用于表示职工的信息,如姓名、工龄、工资等;函数 input 输入一个职工的有关数据,函数 display 显示这些信息。并设计一个使用这两个函数的程序,该程序可以处理 10 个职工的数据。

8.11 某商店的各种商品有下列信息:商品名称、商品价格、商品质量级别等级、商品属哪个货柜管理、商品的库存量。对商品的操作有:商品进库、商品出库、商品调价、增加和删除商品目录。要求用结构编写一个程序,完成要求的操作。

8.12 使用枚举类型定义月份,并编写一个程序,根据用户输入的年份,输出该年各月的天数。

8.13 定义一个描述学生成绩等级的枚举类型{A,B,C,D,E},成绩等级与分数段的对应关系为 A:90~100,B:80~89,C:70~79,D:60~69,E:0~59。在主函数中定义全班学生成绩枚举类型数组,输入全班学生的分数,并转换成等级赋给枚举数组元素。最后输出全班学生的成绩等级。

第 9 章 链　表

【学习内容】

本章介绍 C++ 的链表这一基本数据结构。主要内容包括：

◆ 自引用结构、链表的概念。

◆ 内存的动态分配和释放。

◆ 单向链表的定义与操作。

◆ 双向链表的定义与操作。

【学习目标】

◆ 理解自引用结构和链表的含义。

◆ 掌握通过动态分配和释放内存来建立和维护链表。

通过前面 8 章的学习，已经掌握了 C++ 基本的程序设计机制。特别是在学习了指针和结构之后，可以利用这两种机制设计出非常灵活的数据结构。数据结构是由简单类型的数据构造复合类型数据的方法和表示。数据结构是程序设计的重要方法，有专门的课程和教材进行介绍。本章仅介绍一种最基本数据结构——链表，帮助读者理解链表的定义和实现方法，为进一步学习其他数据结构打下一个基础。

9.1　链表的基本概念

如果需要保存多个同一类型的结构的信息，可以采取结构数组的方式进行管理。但是，与其他静态分配的数据一样，在声明结构数组时，必须将数组的元素数目设定成实际处理时可能需要的最大的数目，这就有可能造成内存空间的浪费。为此，希望能够对于结构也采取动态分配策略，提高内存空间的利用效率。

本章将综合利用前面学习的动态内存分配、结构和指针，建立一种非常有用的动态数据结构——链表。链表是通过指针链接在一起的一组数据项(称为“结点”)。链表的每个结点都是一个同类型的结构，其中有的成员用于存储结点的信息，即用户实际需要的数据；有的成员是指向同类型结构的指针变量，指向了链表中的其他结点。只有一个指针域的链表称为单向链表，有两个指针域的链表称为双向链表。

与结构数组相比较，用链表来组织结构数据有许多优点。在数据元素的数目不可预知时，使用链表是合适的。链表是动态的，所以可在需要的时候增加和减小其长度。而数组是

在编译时分配内存的,数组的大小是不可改变的;用数组的方法必须占用一块连续的内存区域。而使用动态分配时,每个结点之间可以是不连续的(结点内是连续的)。此外,由于结点之间的联系是通过指针实现的,所以对于结点数据的增加和删除也比使用数组更加方便。

9.2 单向链表

9.2.1 单向链表的定义

单向链表中,结点结构中只有一个指针域,用于存放下一个结点的首地址。这样,在第一个结点的指针域内存入第二个结点的首地址,在第二个结点的指针域内又存放第三个结点的首地址,如此串联下去直到最后一个结点。最后一个结点因无后续结点连接,其指针域可置为NULL。

下面是一个单向链表的结点的类型的声明:

```
struct node
{
    int data;
    node * next;
};
```

结构的成员不能为本结构自身的实例。但结构可以包含一个指向本结构的指针成员,如结构node中的next成员可以指向下一个node结构,这种指针称为后继指针。包含指向本结构的指针的结构称为自引用结构。链表的结点大都是自引用结构类型。

单向链表通常有一个指针作为链表头(又称链首指针),该指针指向链表的第一个结点。后续的结点是通过前驱结点中的指针域访问的。通常,单向链表的最后一个结点中的指针域被设置为NULL(表示链尾)。图9-1给出了一个由node构成的单向链表。

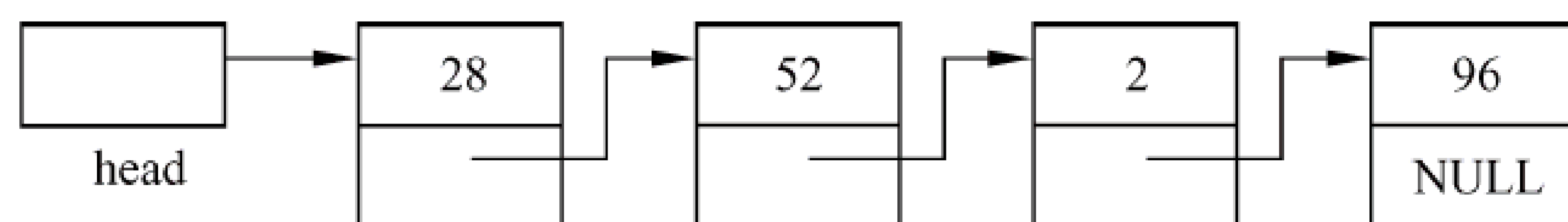


图9-1 单向链表示例

图9-1中,head是链首指针变量,存放有第一个结点的首地址,其声明为:

```
node * head;
```

链表中的每个结点都是node结构类型的,包括两个成员:一个成员存放实际的数据,这里是一个整数;另一个成员为指针,存放下一结点的首地址。单向链表也可以头尾相链接构成单向循环链表。

实际上,链表中的结点可以根据需要保存各种类型的数据。下面给出了一个存放学生信息的链表结点的声明:

```
struct student
{
    int num;
```

```
    char name[20];
    char sex;
    float score;
    student * next;
};
```

前4个成员项组成了数据域，最后一个成员next构成指针域，它是一个指向student类型结构的指针变量。

第7章中介绍了动态内存分配方法：通过malloc和free、new和delete实现内存空间的申请和释放。其中动态申请内存的方法是：

```
TypeName * p=(TypeName*)malloc(sizeof(TypeName));
```

或者

```
TypeName * p=new TypeName;
```

其中，TypeName是任何类型名。如果TypeName是一个已定义的结构名，如student，就可以动态建立由student结构的实例构成的链表。

9.2.2　单向链表的操作

对单向链表的基本操作主要有：建立链表、遍历链表，以及在链表中插入或删除结点等。本节为简化讨论而又不失一般性，以前面定义的node结构作为构成链表的结点。

1. 建立单向链表

建立单向链表的工作是把多个结点连接在一起。首先声明一个链首指针变量head，使它指向链表的第一个结点；然后，通过使用动态分配内存的方法建立每个结点，并将这些结点依次链入链表的尾端，使得每个结点中的指针指向下一个结点，末尾结点中的后继指针置为空指针。

【例9-1】　编写一个建立链表的程序，要求依次读入n个整数(n由用户输入)，每读入一个整数，就存入新生成的结点，并将该结点增加到链表的末尾。

```
//ex9_1.cpp:建立链表
#include<iostream.h>

struct node
{
    int data;
    node * next;
};

node * createList(int n);
node * createListInvert(int n);

int main()
{
    int n;
    node * listHead=NULL;
```

```
    cout<<"Please enter the number of nodes: ";
    cin>>n;
    if (n>0)
        listHead=createList(n);
    return 0;
}
node * createList(int n)
{
    node * temp=NULL, * tail=NULL, * head=NULL;
    int num;
    //读入第一个数据,并建立第一个结点
    cin>>num;
    head=new node;                          //为新结点动态分配内存
    if (head==NULL)                         //内存分配不成功
    {
        cout<<"Not memory available!";
        return NULL;
    }
    else
    {
        head->data=num;
        head->next=NULL;
        tail=head;
    }
    //读入后续数据,并存入新的结点,链接到链表的末尾
    for (int i=0; i<n-1; i++)
    {
        cin>>num;
        temp=new node;                      //为新结点动态分配内存
        if (temp==NULL)                     //内存分配不成功
        {
            cout<<"Not memory available!";
            return head;
        }
        else
        {
            temp->data=num;
            temp->next=NULL;
            tail->next=temp;
            tail=temp;
        }
    }
    return head;
}
```

本例中 createList 函数实现了如何建立 n 个结点的链表。建立第一个结点时，头指针 head 指向它。temp 指向新建的结点，tail 指向链表的最后一个结点。语句

```
tail->next=temp
```

把新建结点连接在 tail 所指结点的后面，而语句

```
tail=temp
```

让 tail 指向新的链表末尾结点。图 9-2 给出了 createList 在输入 4 个整数时(依次分别是 1、2、3、4)，建立链表的过程。

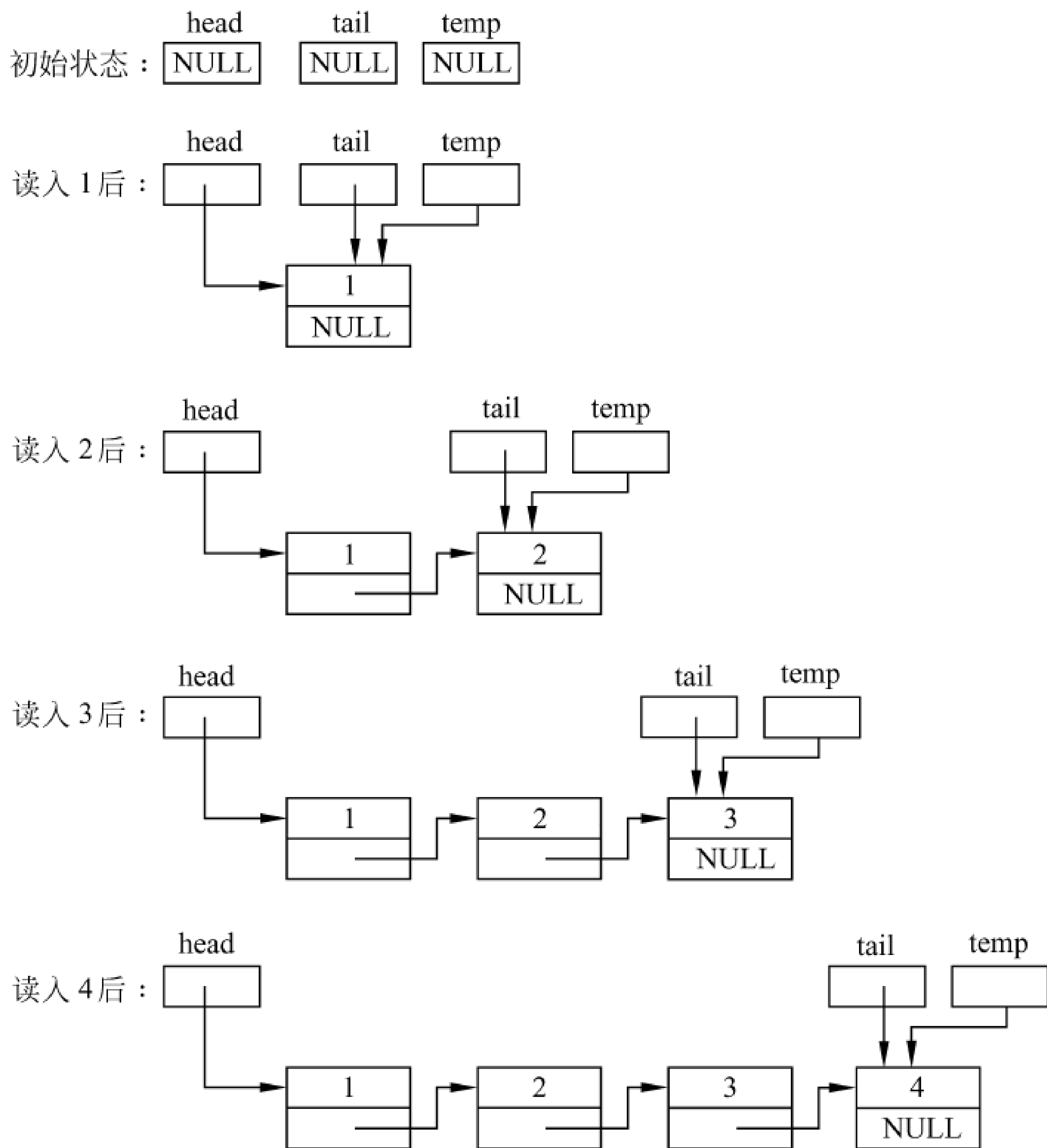


图 9-2　createList 函数建立单向链表的过程

可以看出，在链表建立的过程中，先读入的数据更靠近链表的表头，后读入的数据更靠近链表的表尾。有时也可能需要按照与输入相反的顺序建立链表，即先读入的数据更靠近链表的末尾，后读入的数据更靠近链表的表头。为了实现这一功能，设计了下面的另一个函数 createListInvert。

```
node * createListInvert(int n)
{
    node * temp=NULL, * head=NULL;
    int num;
    //读入第一个数据,并建立第一个结点
```

```
        cin>>num;
        head=new node;                        //为新结点动态分配内存
        if (head==NULL)                       //内存分配不成功
        {
            cout<<"Not memory available!";
            return NULL;
        }
        else
        {
            head->data=num;
            head->next=NULL;
        }
        //读入后续数据,并存入新的结点,链接到链表的末尾
        for (int i=0; i<n-1; i++)
        {
            cin>>num;
            temp=new node;                    //为新结点动态分配内存
            if (temp==NULL)                   //内存分配不成功
            {
                cout<<"Not memory available!";
                return head;
            }
            else
            {
                temp->data=num;
                temp->next=head;
                head=temp;
            }
        }
        return head;
    }
```

createListInvert 函数按照与输入相反的顺序建立 n 个结点的链表。与 createList 函数一样,建立第一个结点时,头指针 head 指向它。temp 指向新建的结点。语句

```
temp->next=head
```

把 head 所指的当前的链首结点链接到新建结点的后面,而语句

```
head=temp
```

将 head 更新为指向新建的结点,即新的链首。与 createList 函数比较,createListInvert 函数少用了一个指针 tail。图 9-3 给出了 createListInvert 在输入 4 个整数时(依次分别是 1、2、3、4),建立链表的过程。

2. 遍历链表

例 9-1 建立了链表,但并没有看到建立的链表的内容,这需要通过遍历链表这一操作来

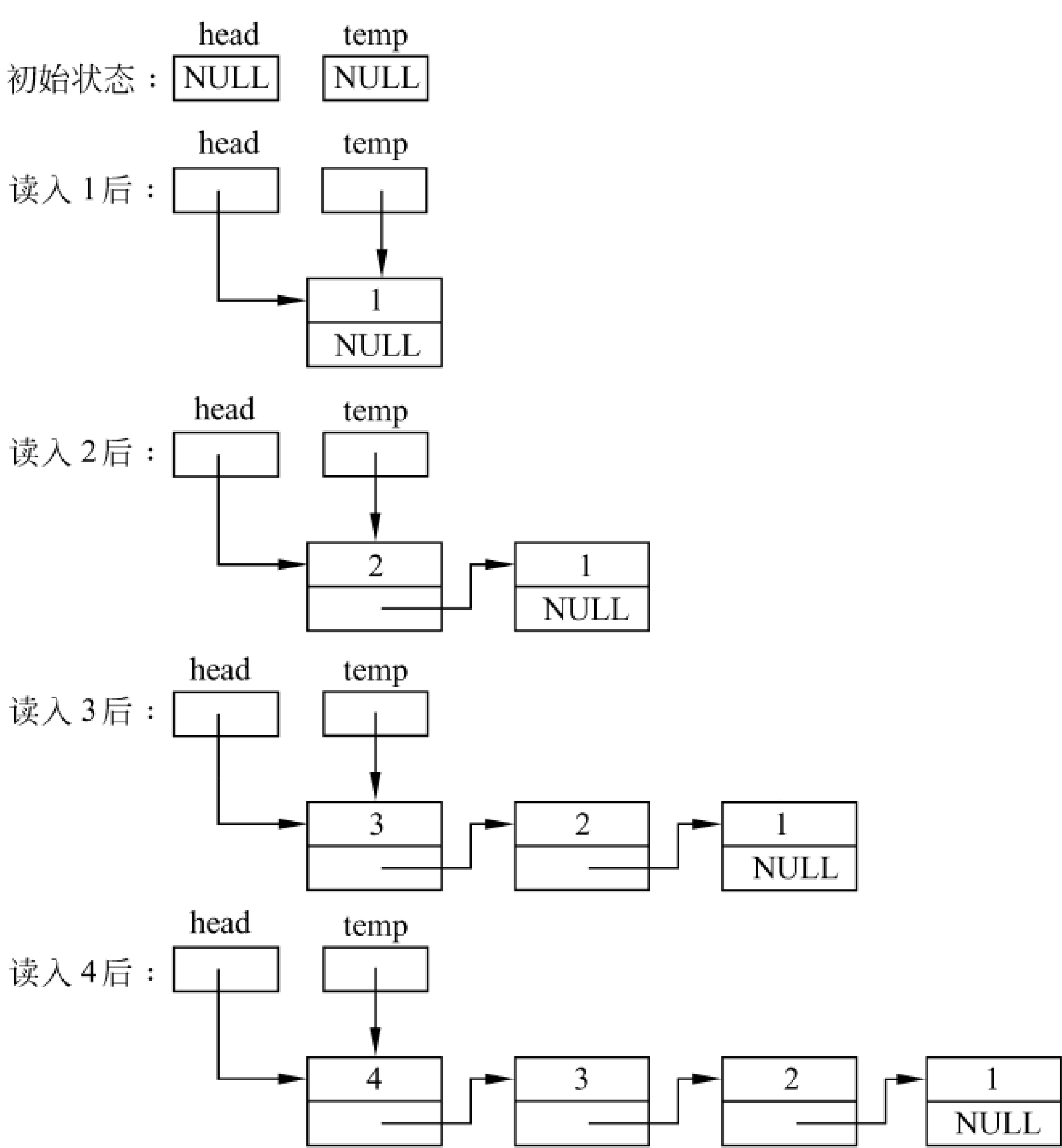


图9-3　createListInvert函数建立单向链表的过程

实现。遍历链表是从链首开始，依次访问（包括输出、修改）链表中的每个结点的信息。链首指针head指向第一个结点，存取它的成员很容易。例如，通过

```
head->data=15;
```

就可以给第一个结点的data成员赋值。

利用第一个结点的后继指针next可以访问到第二个结点，进而可以访问该结点的成员。例如：

```
head->next->data
```

和

```
head->next->next
```

即表示第二个结点中相应的成员。

如果链表中有多个结点，模仿上面的方法访问链表的后继结点显然不大方便。为此，可以声明一个node *类型的指针变量curNode，用于指向当前结点，首先让curNode指向第一个结点，即

```
curNode=head;
```

通过

```
curNode=curNode->next;
```

使得指针 curNode 指向当前结点的后继结点,如果反复执行这一赋值直到 curNode 变成 NULL,就可以遍历链表的所有结点。

因此,遍历一个链表的基本过程如下:

```
node * curNode=head;
while (curNode)
    curNode=curNode->next;
```

【例 9-2】 编写一个函数,以链表形式输出例 9-1 所建立的链表中各结点的 data 成员的值。

设计思想:设计函数 outputList 实现该功能,该函数有一个类型为 node * 的参数 head,指向被遍历输出的链表的链首。输出时,用—>来模拟链表的链接关系。

```
void outputList(node * head)
{
    cout<<"List: ";
    node * curNode=head;
    while (curNode)
    {
        cout<<curNode->data;
        if (curNode ->next)
            cout<<" ->";
        curNode=curNode ->next;
    }
    cout<<endl;
    return;
}
```

以遍历链表的功能为基础,可以进一步实现在链表中查找特定结点的功能。

【例 9-3】 编写一个函数,能够在例 9-1 所建立的链表中查找包含指定的整数的结点。如果找到,则返回指向该结点的指针;否则,返回空指针 NULL。

设计思想:用函数 findData 实现该功能,该函数有两个参数:整型参数 n,表示被查找的整数;类型为 node * 的参数 head,指向被遍历搜索的链表的链首。该函数在遍历结点的过程检查访问的结点是否包含了被查找的整数。

```
node * findData(int n, node * head)
{
    node * curNode=head;
    while (curNode)
    {
        if (curNode->data==n)
        {
            //找到 n
            cout<<"Find "<<n<<" in the list."<<endl;
            return curNode;
```

```
        }
        curNode=curNode->next;
    }
    //未找到 n
    cout<<"Can't find "<<n<<" in the list."<<endl;
    return NULL;
}
```

3. 在链表中插入和删除一个结点

链表的优点之一就是灵活性好，适合数据的动态变化。实际上，在链表中插入或删除一个结点也比在结构数组中增加或减少一个元素要容易得多，因为后者需要通过大量的结构赋值来移动元素以增加或消除元素的空位。

在链表中增加和删除一个结点，要保持链表的连续性。增加一个结点时，插入的结点要与其前后的结点建立指针链接；删除一个结点时，删除的结点的前驱结点和后继结点要重新建立链接。总之，都需要找到插入或删除点的前驱结点和后继结点。但是，对于单向链表来说，由于每个结点仅有一个后继指针，指向后继结点，没有直接指向前驱结点的指针，因而不能直接引用前驱结点。所以，需要在遍历链表定位插入点或删除点的同时，注意记录前驱结点和后继结点的位置。

在链表中结点 a 之后插入结点 c 的过程如下。

(1) 指针 cptr 指向结点 c，aptr 指向结点 a。

(2) 把 a 后继结点的地址赋给结点 c 的后继指针。

```
cptr->next=aptr->next;
```

(3) 把 c 的地址赋给结点 a 的后继指针。

```
aptr->next=cptr;
```

注意，第(2)步和第(3)步不能颠倒。此外，如果需要在链首结点之前插入新的结点，只需要将新结点的后继指针指向原来的链首结点即可。

从链表中删除一个结点 c 的过程如下。

(1) 在链表中查找要删除的结点 c，用指针 cptr 指向结点 c。

(2) 如果 c 有前驱结点(设为 d，用指针 dptr 指向 d)，则将 d 的后继指针指向 c 的后继结点。

```
dptr->next=cptr->next;
```

(3) 释放 c 占用的空间。

【例 9-4】 编写一个链表的插入函数，将输入的整数按照从小到大的顺序插入到链表中合适位置(假设链表中的数据已经按照从小到大的顺序排列好了)。

设计思想：设计函数 insertData 来实现该功能。该函数有两个参数：一个是整型参数 n，表示被插入的整数；另一个是类型为 node * 的参数 head，指向被遍历搜索的链表的链首。该函数首先遍历链表，找到插入的位置，然后将新建的结点插入链表中，并返回更新后的链表的链首指针。

```
node * insertData(int n, node * head)
{
    node * curNode=head;              //指向当前结点
    node * preNode=NULL;              //指向当前结点的前驱结点
    node * newNode=NULL;              //指向新建结点
    //寻找插入位置
    while ((curNode!=NULL) && (curNode->data<n))
    {
        preNode=curNode;              //当前结点变为前驱结点
        curNode=curNode->next;        //当前结点的后继结点变为当前结点
    }
    newNode=new node;                 //为新结点动态分配内存
    if (newNode==NULL)
    {//内存分配不成功
        cout<<"No memory available!";
        return head;                  //返回原来的链表头指针,原来的链表未发生改变
    }
    //内存分配成功
    newNode->data=n;
    if (preNode==NULL)
    {
        //插入位置是链表头,新建结点插入到链首结点之前
        newNode->next=curNode;        //将新结点的后继指针指向链表原来的第一个结点
        return newNode;               //返回指向新结点的指针作为新链表的链首指针
    }
    else
    {
        //插入位置是链表中部,新建结点插入到前驱结点之后、当前结点之前
        //或者插入位置是链表尾,这时当前结点 curNode 为 NULL
        //将前驱结点的后继指针指向新建结点
        preNode->next=newNode;
        //将新建结点的后继指针指向当前结点 curNode
        newNode->next=curNode;
        return head;                  //返回原来的链首指针作为新链表的链首指针
    }
}
```

类似地,可以实现在链表中删除一个结点的功能。

【例 9-5】 编写一个函数,能够根据指定的整数,在前面程序所建立的链表中找到包含该整数的结点并删除之。

设计思想:设计函数 deleteData 来实现该功能,该函数与 insertData 函数类似,也有两个参数:一个是整型参数 n,表示被删除的整数;另一个是类型为 node * 的参数 head,指向被遍历搜索的链表的链首。该函数同样先遍历链表,找到拟删除的结点,然后将该结点从链表中移出并释放,返回更新后的链表的链首指针。

```
node * deleteData(int n, node * head)
{
    node * curNode=head;                //指向当前结点
    node * preNode=NULL;                //指向当前结点的前驱结点
    if (head==NULL)                     //如果链表为空,则直接返回
        return NULL;
    while ((curNode!=NULL) && (curNode->data!=n))
    {
        preNode=curNode;                //当前结点变为前驱结点
        curNode=curNode->next;          //当前结点的后继结点变为当前结点
    }
    //未找到 n
    if (curNode==NULL)
    {
        cout<<"Can't find "<<n<<" in the list."<<endl;
        return head;
    }
    if (preNode==NULL)
        //被删除的结点是链首结点
        //head指向链首结点的后继结点,将链首结点从链表中移出
        head=head->next;
    else
        //被删除的结点在链表中部(包括链表尾)
        //将当前结点的前驱结点的后继指针指向当前结点的后继结点
        //将当前结点从链表中移出
        preNode->next=curNode->next;
    delete curNode;
    return head;                        //返回链首指针
}
```

下面编写一个主程序以调用上面的各个函数。

【例 9-6】 单向链表的综合操作示例。

```
int main()
{
    int n;
    int num;
    node * listHead=NULL;
    cout<<"Please enter the number of nodes: ";
    cin>>n;
    cout<<"Please enter "<<n<<"integers: ";
    for (int i=0; i<n; i++)
    {
```

```
            cin>>num;
            //将num按照从小到大的顺序插入链表中
            listHead=insertData(num, listHead);
        }
        //输出建立的链表
        outputList(listHead);
        //查找结点
        cout<<"\nPlease enter the number to be searched: ";
        cin>>num;
        findData(num, listHead);
        //删除结点
        cout<<"\nPlease enter the number to be deleted: ";
        cin>>num;
        listHead=deleteData(num, listHead);
        //输出删除结点后的链表
        outputList(listHead);
        return 0;
    }
```

主程序的执行结果为:

```
Please enter the number of nodes: 5
Please enter 5 integers: 5 3 1 2 4
List: 1->2->3->4->5
Please enter the number to be searched: 4
Find 4 in the list.
Please enter the number to be deleted: 3
List: 1->2->4->5
```

*9.3 双向链表

9.3.1 双向链表的定义

前面所讨论的是单向链表,每个结点都只有一个指针指向其后继结点。显然,单向链表有利于从链首向链尾遍历,对于每个结点来说,能够很方便地访问其后继结点。但是,这种链表的局限性也是明显的,对于每个结点,访问其前驱结点是困难的,必须借助一个指针专门指向当前结点的前驱结点;而且,对于单向链表,要实现从链尾向链首扫描也不方便。

有时为了提高数据处理的灵活性,需要支持双向访问结点,即从当前结点要能直接访问其前驱结点和后继结点,既要支持从链首向链尾遍历,也要支持从链尾向链首扫描。此时,双向链表就很有用了。双向链表中的结点除了含有数据域外,还包括两个指针域,其中一个指向其前驱结点,另一个指向其后继结点。下面是一个双向链表的结点的类型的声明。

```
struct node
```

```
{
    int data;
    node * next;
    node * pre;
};
```

图 9-4 说明了一个由 node 构成的双向链表。

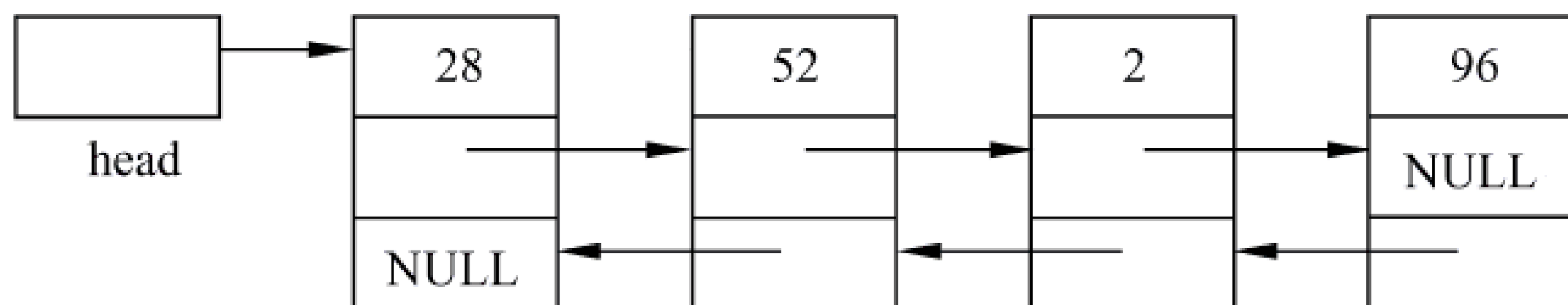


图 9-4　双向链表示例

双向链表一般也由头指针 head 唯一确定。双向链表也可以头尾相链接构成双(向)循环链表。双向链表的最大优点就是在搜索链表的数据时,不但可以往前,也可以向后寻找。同时,在进行结点的插入和删除时也相对直接一些。

9.3.2　双向链表的操作

双向链表和单向链表一样,其基本操作有:建立链表、遍历链表以及在链表中插入或删除结点等。出于同样的考虑,本节是以 9.3.1 节定义的 node 结构作为构成链表的结点。总体说来,在双向链表中,有些操作仅需涉及一个方向的指针,那么它们的算法和单向链表的操作类似。但是,有一些操作,如插入、删除结点时有较大的不同,虽然双向指针为访问前驱和后继结点提供了方便,但在维护链表指针上又略显复杂。

1. 建立双向链表

建立双向链表比建立单向链表稍复杂:如果是将新建的结点链入链尾,需要将原来的链尾结点的后继指针指向新的结点,而新的结点的前驱指针指向原来的链尾结点,新的链尾结点的后继指针置为空指针;如果是将新建的结点链入链首,需要将原来的链首结点的前驱指针指向新的结点,而新的结点的后继指针指向原来的链首结点,新的链首结点的前驱指针置为空指针。

【例 9-7】　编写两个函数,分别按照输入的顺序及其逆序建立双向链表。

设计思想:用 createBidirList 函数实现按照输入的顺序建立 n 个结点的双向链表,用 createBidirListInvert 函数按照与输入相反的顺序建立 n 个结点的双向链表。这两个函数具有相同的参数和返回值类型:整型参数 n 表示拟输入的整数的数目,返回一个 node * 类型的指针,指向构造好的双向链表的链首。

```
node * createBidirList(int n)
{
    node * temp=NULL, * tail=NULL, * head=NULL;
    int num;
    if (n<=0)
        return NULL;
    //读入第一个数据,并建立第一个结点
```

```
    cin>>num;
    head=new node;                          //为新结点动态分配内存
    if (head==NULL)                         //内存分配不成功
    {
        cout<<"No memory available!";
        return NULL;
    }
    else
    {
        head->data=num;
        head->next=NULL;
        head->pre=NULL;
        tail=head;
    }
    //读入后续数据,并存入新的结点,链接到链表的末尾
    for (int i=0; i<n-1; i++)
    {
        cin>>num;
        temp=new node;                      //为新结点动态分配内存
        if (temp==NULL)                     //内存分配不成功
        {
            cout<<"No memory available!";
            return head;
        }
        else
        {
            temp->data=num;
            temp->next=NULL;
            temp->pre=tail;
            tail->next=temp;
            tail=temp;
        }
    }
    return head;
}
node * createBidirListInvert(int n)
{
    node * temp=NULL, * head=NULL;
    int num;
    if (n<=0)
        return NULL;
    //读入第一个数据,并建立第一个结点
    cin>>num;
```

```
    head=new node;                                   //为新结点动态分配内存
    if (head==NULL)                                  //内存分配不成功
    {
        cout<<"Not memory available!";
        return NULL;
    }
    else
    {
        head->data=num;
        head->next=NULL;
        head->pre=NULL;
    }
    //读入后续数据,并存入新的结点,链接到链表的头
    for (int i=0; i<n-1; i++)
    {
        cin>>num;
        temp=new node;                               //为新结点动态分配内存
        if (temp==NULL)                              //内存分配不成功
        {
            cout<<"No memory available!";
            return head;
        }
        else
        {
            temp->data=num;
            temp->next=head;
            temp->pre=NULL;
            head->pre=temp;
            head=temp;
        }
    }
    return head;
}
```

2. 双向链表的遍历

对双向链表进行遍历与单向链表没有差别,如果提供了链首结点,则可以沿着结点后继指针从头至尾进行遍历;如果提供了链尾结点,也可以沿着结点的前驱指针从尾向头进行遍历。

例 9-8 是在构造好的双向链表的基础上,根据输入参数提供的双向链表的链首指针进行遍历,对链表中的数据进行输出,与单向链表的操作基本上一样。例 9-9 则是根据输入参数提供的双向链表的链尾指针进行遍历,对链表中的数据进行查询,与单向链表的相应操作也基本类似。

【例 9-8】 编写一个函数,以链表形式输出双向链表中各结点的 data 成员的值。

设计思想：函数 outputBidirList 实现该功能，该函数有一个类型为 node * 的参数 head，指向被遍历输出的链表的链首。该函数从链首出发，沿着指向后继结点的指针遍历双向链表。输出时，用－＞来模拟链表的后继指针，用＜－来模拟链表的前驱指针。

```
void outputBidirList(node * head)
{
    cout<<"List: ";
    node * curNode=head;
    while (curNode)
    {
        if (curNode->pre)
            cout<<" <- ";
        cout<<curNode->data;
        if (curNode->next)
            cout<<" -> ";
        curNode=curNode->next;
    }
    cout<<endl;
    return;
}
```

【例 9-9】 编写一个函数，能够在双向链表中从链尾出发查找包含指定整数的结点。如果找到，则返回指向该结点的指针；否则，返回空指针 NULL。

设计思想：函数 findData 实现该功能，该函数有两个参数：整型参数 n，表示被查找的整数；类型为 node * 的参数 tail，指向被遍历搜索的双向链表的链尾。该函数从链尾出发，沿着指向前驱结点的指针遍历双向链表。

```
node * findData(int n, node * tail)
{
    node * curNode=tail;
    while (curNode)
    {
        if (curNode->data==n)
        {
            //找到 n
            cout<<"Find "<<n<<" in the list."<<endl;
            return curNode;
        }
        curNode=curNode->pre;
    }
    //未找到 n
    cout<<"Can't find "<<n<<" in the list."<<endl;
    return NULL;
}
```

3. 在双向链表中插入和删除一个结点

与单向链表相比较，在双向链表中插入和删除一个结点，方便之处在于访问插入结点或被删除结点的前驱和后继结点比较方便，勿须借助额外的指针变量。但是，为了保持双向链表的性质，需要维护的指针比单向链表要多。注意，双向链表的链首结点和链尾结点的特殊性：链首结点的前驱指针和链尾结点的后继指针是空的。下面，仍然以按照从小到大的顺序保存整数的双向链表为例，设计在双向链表中插入的函数。

【例 9-10】 编写一个双向链表的插入函数，将输入的整数按照从小到大的顺序插入到链表中合适位置(假设链表中的数据已经按照从小到大的顺序排列好了)。

设计思想：设计一个新的 insertData 函数实现该功能，该函数仍然具有两个参数：一个是整型参数 n，表示被插入的整数；另一个是类型为 node * 的参数 head，指向被遍历搜索的双向链表的链首。该函数遍历链表，找到插入的位置，然后将新建的结点插入链表中，并返回更新后的链表的链首指针。

```
node * insertData(int n, node * head) {
    node * curNode=head;                //指向当前结点
    node * preNode=NULL;                //指向当前结点的前驱结点
    node * newNode=NULL;                //指向新建结点
    //寻找插入位置
    while((curNode !=NULL) && (curNode->data<n))
    {
        preNode=curNode;                //当前结点变为前驱结点
        curNode=curNode->next;          //当前结点的后继结点变为当前结点
    }
    newNode=new node;                   //为新结点动态分配内存
    if (newNode==NULL)
    {
        //内存分配不成功
        cout<<"Not memory available!";
        return head;                    //返回原来的链表头指针，原来的链表未发生改变
    }
    newNode->data=n;
    if (preNode==NULL)
    {
        //插入位置是链表头，新建结点插入到链首结点之前
        newNode->next=curNode;
                                //将新结点的后继指针指向当前结点(链表原来的第一个结点)
        newNode->pre=NULL;              //新结点的前驱指针置为 NULL
        if (curNode !=NULL)             //判断是否为空链
            curNode->pre=newNode;
                                    //当前结点(链表原来的第一个结点)的前驱指针指向新结点
        return newNode;                 //返回指向新结点的指针作为新链表的链首指针
    }
    if (curNode==NULL)
    {
```

```
            //插入位置是链表尾,新建结点插入到 preNode 指向的结点之后
            newNode->pre=preNode;               //新结点的前驱指针指向 preNode
            preNode->next=newNode;              //preNode 的后继指针指向新结点
            newNode->next=NULL;                 //新结点的后继指针置为 NULL
            return head;                        //返回原来的链首指针作为新链表的链首指针
        }
        else
        {
            //插入位置是链表中,新建结点插入到前驱结点之后、当前结点之前
            preNode->next=newNode;              //前驱结点的后继指针指向新结点
            newNode->next=curNode;              //新结点的后继指针指向当前结点
            newNode->pre=preNode;               //新结点的前驱指针指向前驱结点
            curNode->pre=newNode;               //当前结点的前驱指针指向新结点
            return head;                        //返回原来的链首指针作为新链表的链首指针
        }
    }
```

【例 9-11】 编写一个函数,能够根据指定的整数,在前面程序所建立的双向链表中找到包含该整数的结点并删除之。

设计思想:设计函数 deleteData 实现该功能,该函数有两个参数:一个是整型参数 n,表示被删除的整数;另一个是类型为 node * 的参数 head,指向被遍历搜索的链表的链首。该函数遍历链表,找到拟删除的结点,并将该结点从链表中移出并释放,返回更新后的链表的链首指针。

```
node * deleteData(int n, node * head)
{
    node * curNode=head;                        //指向当前结点
    while (curNode)
    {
        if (curNode->data==n)
        {
            //找到 n,删除当前结点
            if (curNode->pre==NULL)
            {
                //被删除的结点是链首结点
                //head 指向链首结点的后继结点,将链首结点从链表中移出
                head=head->next;
                head->pre=NULL;                 //新的链首结点的前驱指针置为 NULL
            }
            else
            {
                //被删除的结点在链表中部(包括链表尾)
                //将当前结点从链表中移出
                //将当前结点的前驱结点的后继指针指向当前结点的后继结点
                curNode->pre->next=curNode->next;
                if (curNode->next!=NULL)
```

```
                //当前结点不是链尾结点
                //将当前结点的后继结点的前驱指针指向当前结点的前驱结点
                curNode->next->pre=curNode->pre;

            }
            delete curNode;
            cout<<"Delete "<<n<<" from the list."<<endl;
            return head;                    //返回链首指针
        }
        curNode=curNode->next;              //当前结点的后继结点变为当前结点
    }
    //未找到 n
    cout<<"Can't find "<<n<<" in the list."<<endl;
    return head;
}
```

下面给出一个综合操作双向链表的主程序，以调用上述操作函数。

【例 9-12】 双向链表的综合操作示例。

```
int main()
{
    int n;
    int num;
    node * listHead=NULL;
    node * listTail=NULL;
    cout<<"Please enter the number of nodes: ";
    cin>>n;
    cout<<"Please enter "<<n<<" integers: ";
    for (int i=0; i<n; i++)
    {
        cin>>num;
        //将 num 按照从小到大的顺序插入链表中
        listHead=insertData(num, listHead);
    }
    //输出建立的链表
    outputBidirList(listHead);
    //查找结点
    cout<<"\nPlease enter the number to be searched: ";
    cin>>num;
    //找到链表尾
    listTail=listHead;
    if (listTail)
        while (listTail->next!=NULL)
```

```
            listTail=listTail->next;
    findData(num, listTail);
    //删除结点
    cout<<"\nPlease enter the number to be deleted: ";
    cin>>num;
    listHead=deleteData(num, listHead);
    //输出删除结点后的链表
    outputBidirList(listHead);
    return 0;
}
```

程序执行结果:

```
Please enter the number of nodes: 5
Please enter 5 integers: 5 3 1 2 4
List: 1->  <-2->  <-3->  <-4->  <-5
Please enter the number to be searched: 4
Find 4 in the list.
Please enter the number to be deleted: 3
Delete 3 from the list.
List: 1->  <-2->  <-4->  <-5
```

习 题 9

9.1 什么是链表?链表的基本操作有哪些?在链表的建立、插入、删除、输出函数中的关键是什么?

9.2 编写一个程序,建立一条单向链表,每个结点包含姓名、学号、英语成绩、数学成绩和 C++ 成绩,并通过链表操作,求出平均分最高和最低的同学并且输出。

9.3 编写一个程序,把两个字符单向链表连接起来。程序包含函数 concatenate,函数以对两个链表的引用作为参数,把第二个链表连接到第一个链表后面。

9.4 一般来说,对于单向链表,要实现从链尾向链首扫描并不容易。能否借助递归方法实现单向链表的从链尾向链首扫描?请编写程序实现这一功能。

9.5 编写一个程序,建立一个单向链表,用一个函数实现将这条链表逆转过来,即将原来的表头变成表尾,原来的表尾变成表头。

9.6 编写一个程序,把两个有序排列的单向整数链表合并成一个有序排列的整数链表。函数 merage 以对两个指针作为参数,这两个指针分别指向两个待合并链表的头结点,函数返回一个指针,它指向合并后的链表的头结点。

9.7 设计一个记录书籍的结构,其成员包括书号、书名、作者、出版社和出版日期。假设一批书籍的记录存放在一个单向链表中,编写函数输出与给定关键字相匹配的所有书籍的记录。例如,设某给定关键字为一个作者的姓名,则将该作者的所有著作都显示出来。

9.8 已知由单向链表中含有 3 类字符的数据元素(如字母字符、数字字符和其他字符),试编写程序构造 3 个以循环链表表示的线性表,使每个表中只含同一类的字符,且利用原表中的结点空间作为这 3 个表的结点空间,头结点可另辟空间。

9.9 已知单向链表中的元素以值递增有序排列,试编写一个程序,删除链表中所有值相同的多余元素(使

得运算后的链表中所有元素的值均不相同)。

9.10　有两个双向链表 a 和 b,设结点中包含学号、姓名。从 a 链表删去与 b 链表中有相同学号的那些结点。

9.11　由 13 个人围成一圈,从第一个人开始顺序报号 1,2,3。凡是报到 3 的人退出圈子,找出最后留在圈子中的人原来的序号。请用双向链表来实现这个游戏,输出每次退出者的顺序号。

9.12　设有一个双向链表,每个结点中除有 pre、data 和 next 3 个域外,还有一个访问频度域 freq,在链表使用之前,其每个结点的 freq 值均初始化为零。请编写一个在该双向链表中查找包含指定的整数的结点的函数,该函数有两个参数:整型参数,表示被查找的整数;链表的表头指针,指向被访问的双向链表的链首。如果找到,则返回指向该结点的指针,并将该结点的域 freq 加 1;否则,返回空指针 NULL。要求,多查找后,此链表中结点保持按访问频度递减的顺序排列,以便使频繁访问的结点总是靠近表头。

第10章 面向对象程序设计基本概念

【学习内容】

本章介绍面向对象程序设计的基本概念。主要内容包括：

◆ 面向对象程序设计方法的产生和发展。

◆ 面向对象程序设计语言。

◆ 面向对象程序设计的特点。

◆ 类和对象的基本概念。

◆ 消息。

【学习目标】

◆ 了解面向对象程序设计方法相对于结构化程序设计方法的优点。

◆ 了解面向对象程序设计语言的特点。

◆ 掌握类、对象和消息在面向对象程序设计中的作用。

◆ 了解面向对象程序的结构。

计算机只是一台机器，计算机语言也只是描述机器应该如何工作的工具。但随着计算机技术的发展，计算机在很多应用中已不仅仅是一台机器，它已能体现和人类似的一些特性，并能在越来越多的领域替代人脑的活动。在某些应用中，计算机已经像一个有自我表述能力和自我处理能力的个体。面向对象程序设计(object oriented programming，OOP)是计算机向具有自我表述和自我处理能力的个体发展中的一项重要技术。

本章首先介绍面向对象程序设计方法的起源及发展历程，然后讨论在面向对象程序设计中，需要采用什么策略来设计自己的系统，具体内容包括类、对象的概念、消息，以及面向对象程序的结构和面向对象程序设计的特点等。

10.1 面向对象语言和面向对象方法

程序设计语言的发展经历了机器语言、汇编语言和高级语言等阶段，总的趋势是描述手段越来越高级，越来越接近自然语言或数学语言，越来越贴近客观世界本身。在高级语言程序设计方法的发展过程中，经历了结构化程序设计和面向对象程序设计两个阶段。C++ 就是在 C 结构化程序设计语言的基础上发展而来的，前面的章节详细介绍了 C++ 对结构化程序设计方法的支持，从本章开始，将系统阐述使用 C++ 进行面向对象程序设计的方法。

面向对象程序设计方法起源于20世纪60年代末的Simula67语言。Simula67可以说是面向对象语言的鼻祖,它将结构化程序设计语言ALGOL 60中块结构的概念向前推进了一大步,提出了对象的概念。对象代表待处理问题中的实体,在处理问题的过程中,一个对象可以采用某种方法与其他对象通信。从概念上讲,对象是包含数据和处理这些数据的操作的程序单元。Simula67语言中也包含了类和继承的概念,类用来描述特性相同或相近的一组对象的结构和行为,继承则可以将多个类组织成层次结构,进而实现对数据和操作的共享。

20世纪70年代末80年代初,Alan Kay、Dan Ingalls、Ted Kaehler和Adele Goldberg等共同开发了面向对象语言Smalltalk。Smalltalk是第二个面向对象的程序设计语言和第一个真正的集成开发环境(IDE)。Smalltalk引入了Simula67语言的许多面向对象的特征,包括类和继承等,其信息的隐藏也更加严格。作为一种集成开发环境,Smalltalk附带有一个庞大的、标准的类库,极大地提高了使用Smalltalk编写程序的效率。Smalltalk还是第一个支持MVC(model-view-controller)设计模式的应用开发环境。Smalltalk作为一种成功的面向对象语言,对后来的很多程序设计语言的产生起到了重要的推动作用,如C++、Java、CLOS(面向对象的Lisp)、面向对象的Pascal和面向对象的Basic等。

在Smalltalk之后,面向对象方法开始为人们注目,特别是C++的推出,使面向对象语言在工业界广为人知。之后,面向对象语言被分为两大阵营:一是以Smalltalk、Eiffel、Java等为代表的纯粹的面向对象语言;另一个则是以C++和CLOS为代表的混合型面向对象语言。前者更强调软件开发的探索性和系统的原型化开发;后者则是对现有程序设计语言的扩充,强调运行时的时间效率,并且已经被工业界广泛接受。

和Smalltalk同时期出现的Ada语言也是一种支持对象的非常重要的语言。Ada语言中的抽象机制是包(Smalltalk、C++、Java中提供的抽象机制都是类)。它支持数据抽象类型、函数和运算符重载以及多态性等面向对象的机制。但是,Ada语言并不全面支持继承,因此人们常称它为一种基于对象的语言,而不是面向对象的语言。

在计算机科学的发展过程中,程序设计语言的发展总是推动了程序设计方法的进步,甚至带来了软件开发方法的革命。以Pascal和C为代表的结构化程序设计语言使得结构化方法不断发展,形成了系统的、支持软件开发整个过程的结构化分析、设计和程序设计方法。结构化方法引入了工程和结构化思想,结束了以前软件开发的混乱状态,使大型软件的开发技术和过程都得到了极大的改善。但是,随着用户功能需求的增多,软件变得越来越庞大、复杂,程序的维护、修改成为整个软件开发过程中非常繁杂的工作,传统的结构化程序设计方法受到了严峻的考验。在这个过程中,面向对象方法则越来越显示其优势,并逐渐成为主流。

面向对象方法直接将问题的求解映射到对问题的认识上,提供了一种有目的地将系统分解为模块的策略,并且将软件设计决策与客观世界的认识相匹配。面向对象程序设计允许将问题分解为一系列的实体(对象),然后围绕这些实体建立数据和对数据进行处理的方法(在C++中就是函数)。当程序越来越大时,使用结构化程序设计方法进行程序设计会使工作变得拙劣而混乱,而一个支持面向对象程序设计概念的程序设计语言则可以让问题变得简单和自然。

在一个面向对象的系统中,对象是程序运行时的基本实体。它可以用来表示一个人、一

个企业、一张桌子或者其他任何需要被程序处理的东西。它也可以用来表示用户定义的数据,如一个向量、一个复数、一个数组等。在面向对象程序设计中,问题的分析一般以对象及对象间的自然联系为依据。当一个程序运行时,对象之间通过互发消息来相互作用。例如,图书馆管理系统的程序中可能包含了一组 reader(读者)对象和一组 librarian(图书管理员)对象,reader 对象可能会向某个 librarian 对象发送一个消息,要求查找某一本书,librarian 对象收到该消息后会执行相应的查找动作,并将查询结果交给相应的 reader 对象。每个对象都包含了一定的数据及操作这些数据的代码。在面向对象方法中,消息是对象之间相互作用的唯一途径,即使不了解彼此的数据和代码的细节,对象之间依然可以通过发送消息和响应消息来相互沟通。即当某个对象需要与其他对象通信时,只需了解其他对象能够接收消息的类型及响应消息后返回的数据,而不必关心其他对象处理消息的具体细节。

面向对象方法同样包括了一系列支持整个软件开发过程的方法,如面向对象需求分析方法、面向对象的软件设计方法和面向对象程序设计方法等。面向对象程序设计方法克服了结构化程序设计方法中的许多缺点。例如,在结构化程序设计方法中,全局数据被系统所有函数共享,这导致程序员很难控制对数据的访问,数据有可能会被不相关的函数意外修改。而在面向对象程序设计中,数据被看做程序开发中的基本元素,不允许它们在系统中自由流动,它将数据和对这些数据的操作紧密联结在一起,并保护数据不会被不相关的代码意外修改。

面向对象的程序设计方法具有以下一些特性。

(1) 程序设计的重点在于数据而不是函数,函数的设计是为数据服务的。

(2) 程序被划分为对象。

(3) 对象之间通过相互发送和响应消息来协同完成程序的功能。

(4) 数据结构的设计是为了体现对象的特性,大多数对象的定义以数据为中心。

(5) 函数作为对某个对象数据的操作,与数据结构紧密地结合在一起。

(6) 数据被隐藏起来,不能为外部程序直接访问。

(7) 新的数据和函数可以在需要的时候方便地添加进来。

(8) 在程序设计过程中遵循自底向上(bottom up)的设计方法。

10.2 类、对象和消息

类、对象和消息是面向对象程序设计的基本要素。进行面向对象程序设计的主要任务是:首先对实际问题加以分析,分辨并抽取出问题中的类和对象,然后设计相应的类,并根据这些自定义或预定义类创建对象,通过这些对象之间的协同工作(发送和响应消息),共同完成程序运行的任务。

10.2.1 类和对象

怎样理解类和对象,它们之间又是什么关系呢?

人们所处的世界是由一个个对象组成的,每个学生、每个班级、每个学校、每个国家、每棵树、每本书、每部汽车都是一个个具体的对象。这些对象都具有一定的属性和行为。例如,学生张三的姓名是“张三”,身高 185cm;李四的姓名是“李四”,身高 160cm。张三的思维

严谨,而李四的思维活跃……。在面向对象程序设计中,对象是现实世界中个体或事物的抽象表示,是属性和相关操作的封装。所谓属性是对象的性质,属性的取值决定了对象所有可能的状态,而操作是对象可以展现的外部服务。

类是某些对象共同特征的表示。类是创建对象的模板,对象是类的实例。类描述对象属性的名称和类型以及方法的实现途径。类的所有实例(对象)具有相同的属性名称及类型、相同的方法、相同的消息响应方式。

可以把类理解为一个抽象的概念,它描述了一类事物都具有的属性和行为。在面向对象方法中,类的属性也称为数据成员,类的行为也称为操作、方法或成员函数。例如,通过“人”这个类,可以说明每个“人”都共同具有的一些属性和行为,如“人”有姓名、身高、体重,有五官和四肢,能交流、能思维等。看下面这些问题:

“人”有姓名,那么“人”的姓名是什么?

“人”有身高,那么“人”的身高是什么?

这些问题无法回答,因为“人”不是一个具体的对象,它只是一个抽象的概念、一个模板,它描述了某一类事物应该具有的属性和行为,但并未给出这些属性的值。作为这一类事物的实例的每一个具体的个体,才能拥有具体的属性取值和行为方式,并且它们的属性的取值各不相同,行为的内容也各异,这些具体的个体就是对象。“人”这个类的实例就是一个个具体的人,如张三和李四,他们都有姓名、身高、体重等属性以及交流和思维等行为,但他们的属性的具体取值可能不同,行为方式也各有特点。类和对象之间的关系可以理解为抽象和具体、类别和实例的关系。

前面说过,面向对象程序设计的主要任务是设计类。但是,类只是一个抽象的概念,设计这些抽象的东西有什么用呢?它们甚至都没有自己的属性值。设计类的目的就是为了创建需要的对象,而这些对象就是面向对象程序中完成各种功能的实体,就像结构化程序设计中的函数一样。

下面的程序定义了用C++定义的读者类Reader。

```
//定义读者类 Reader
class Reader {
public:
    Reader();                               //构造函数
    int register(char * name);              //注册
    int borrowBook(int bookNo);             //借书
    int returnBook(int bookNo);             //还书
private:
    char * name;                            //姓名
    char * certifNo;                        //借书证号
};
```

上面的读者类Reader以关键字class开头,接下来是类名Reader和类的体(花括号括起来的部分),并以分号结束。类Reader的定义分为两个部分,其中在关键字public后面定义的是类Reader的公有成员,类的公有成员可以被外界程序直接访问,因此也称为类的接口。类Reader有4个公有成员函数:Reader、register、borrowBook和returnBook。其中与类同名的成员函数称为类的构造函数,构造函数没有返回类型,专门用来在创建对象时初始

化对象的数据成员。外界可以直接使用这些公有成员函数。关键字 private 后面定义的是类的私有成员,类 Reader 包含两个私有数据成员,分别为 name(姓名)和 certifNo(借书证号)。对私有数据成员的访问是受限的,它们只能被类的成员函数和友元访问。类的数据成员一般都放在 private 部分,可以防止外界程序的非法访问。对于成员函数来说,为外界提供服务的一般放在 public 部分,而不直接为外界提供服务,只是用于支持类的其他成员函数可以放在 private 部分。

定义好的类就是一种新的自定义类型,程序员可以用它来声明变量、指针、数组等。一般情况下,称用类声明的变量为对象。例如:

```
Reader zhang;                      //声明 Reader 类的对象 zhang
Reader readers[1000];              //声明 Reader 类的数组 readers
Reader * rPtr= &zhang;             //声明指向 Reader 类型的指针 rPtr
Reader &ref= zhang;                //声明 Reader 类型的引用
```

可以看出,类名 Reader 作为新的数据类型而存在,程序员可以像使用预定义类型(如 int、float)那样来使用 Reader。一个类可以有许多实例对象,这些对象各自拥有自己的数据空间,互不干扰。类的定义描述了其所有对象的共性,类 Reader 的所有对象都拥有自己的姓名和借书证号,并且可以响应 3 种消息:注册、借书和还书。

C++ 允许程序员建立新的类,这也是为什么说 C++ 是一种可扩展语言的理由之一。

程序设计风格提示: *类名要用能够反映该类的意义的标识符充当,首字符大写,如果用多个单词来描述类名,每个单词的首字符大写,其他字符小写。类的数据成员和成员函数的命名与一般变量和函数的命名要求相同。*

10.2.2 消息

面向对象的程序大多包含许多对象,因此使用面向对象语言进行程序设计的一般步骤如下。

① 创建类,在类中定义数据(属性)和函数(操作)。

② 声明(创建)类的对象。

③ 组织协调各对象来共同完成程序任务。

拥有各种不同功能的对象之后,接下来的任务就是组织这些对象进行工作。这就像一个公司在招聘到不同的人才(如市场人员、财务人员、技术人员、售后服务人员等)之后开始工作一样,这些工作需要公司内部不同人员之间的通力合作才能完成。对于面向对象程序设计来说,也需要建立对象之间的通信和交互机制。

在结构化程序中,程序之间的相互作用是通过函数调用实现的,而在面向对象程序中,程序之间的相互作用是通过对象之间的相互发消息来实现的。对象之间通过消息相互沟通,类似于人与人之间的信息传递。这种消息机制使得面向对象程序设计方法更容易模拟真实世界。

在面向对象程序中,如果两个对象 A 和 B 之间需要交互,对象 A 就向对象 B 发送一个与本次交互相关的特定消息(该消息中包括了对象 A 给对象 B 的参数),对象 B 接收到该消息后会作出相应的响应,即执行一个与该消息对应的成员函数来完成一系列的操作,并将操作的结果返回给对象 A。对于某个对象来说,向该对象发消息就是请求该对象完成某种功

能。为了实现上的方便，在许多面向对象程序设计语言中，都将发送消息和执行响应消息的处理函数合二为一。例如，在C++中，向对象发送消息被处理成调用对象的某个成员函数，调用对象的成员函数可以使用对象名(或对象的引用)加点操作符或指向对象的类指针加箭头操作符的方式进行。沿用10.2.1节的例子：

```
zhang.borrowBook(12);
rPtr->borrowBook(12);
ref.borrowBook(12);
```

上面3条语句都表示向对象zhang发送消息borrowBook(参数是12)，其语义相同：让对象zhang借序号为12的书，对象zhang接收到该消息后，执行与该消息对应的(与消息名相同的)公有成员函数borrowBook(带上参数12)，并将该函数的返回结果(一个整数)返回给发送消息的程序。在C++中，这些复杂的过程被简化为调用对象zhang的成员函数borrowBook，其中zhang是接收消息的对象，borrowBook是要调用的函数名，括号里是函数参数。在后面的章节中，将不加区分地使用“给对象发消息”和“调用对象的成员函数”这两种说法，它们表示同一个意思。

消息机制具有下面3个特性。

(1) 同一个对象可以接收不同形式的多个消息，并产生不同的响应。如对象zhang可接收注册、借书、还书3种消息。

(2) 给不同对象发送同一形式的消息会产生不同的效果。例如，给对象zhang发送消息borrowBook表示读者zhang借书，给对象li发送消息borrowBook则表示读者li借书，产生的结果不同。

(3) 对消息的响应并非必需的。对象不需要总是去响应消息，它也可以忽略某些消息。

对象之间是完全平等的，每一个对象都可以向其他对象发送消息，让其他对象为自己服务。就像在结构化程序中，函数之间可以相互调用一样。面向对象程序设计的任务就变成了如何组织好这些对象之间的关系，以完成各种复杂的任务。

10.3 面向对象程序设计的特点

面向对象程序设计的基本特性是封装、继承和多态性。一个程序设计语言必须支持这几个基本特性才能被称为面向对象的。

1. 抽象和封装

抽象是指提取和表现事物的核心特性而不描述背景细节的行为。面向对象程序设计是通过类的定义进行抽象的，类的定义是对实体的抽象和描述，说明一类对象共同具有的属性名称、类型及方法、操作的实现途径。类把数据及与数据相关的操作组织在一个单独的程序单元中，这种机制称为封装。封装机制是结构化程序设计方法难以支持的。

在结构化程序设计过程中，程序的各功能模块和数据之间的关系是由程序员在自己的头脑(或文档)中保持，在程序中这种约束关系没有显式地体现出来。随着将来系统功能的修改或扩充，程序结构可能会变得越来越差，难以维护。

而在面向对象程序设计中，数据及其相关的操作被封装在类或对象中，程序语言直接支

持对这种相关性的描述和保护。这样,不仅结构清晰,而且系统各组成部分(对象)之间的独立性好,接口关系简单,交互途径单一——只能通过消息机制这一途径进行通信,这些特点都有利于系统功能的扩充和修改。类的封装机制将数据和代码捆绑在一起,避免了外界的干扰和不确定性。类可以用来创建对象。简单地说,一个对象就是一个封装了数据和操作这些数据的代码的逻辑实体。

2. 数据隐藏和访问机制

对象被定义为一系列抽象的属性(如名称、尺寸、质量)以及操作这些属性的函数。类则封装了即将被创建的对象的所有属性和操作。C++中,类是设计抽象数据类型的有效手段。

数据封装带来了另一个好处,即信息隐藏。封装的数据不能被外界直接访问,只能被同一个类中的函数访问。类的公有属性和函数提供了对象和外界程序之间的所有接口。这种隐藏实现细节、避免数据被外界程序直接访问的机制称为"信息隐藏"。

在一个对象内部,某些代码和(或)某些数据可以是私有的,私有成员不能被外界访问。通过这种方式,对象对内部数据提供了不同级别的保护,以防止系统中无关的程序意外地改变或错误地使用对象的私有部分。

3. 继承

面向对象程序设计的核心是类的设计,通过继承机制,可以对已有的类进行扩充,定义出满足自己需要的新类。继承允许从现有的类出发建立新类,新类继承了现有类的属性和行为,并且可以根据自己的需要修改和扩充这些属性和行为。在类的继承中,被继承的类称为基类,新定义的类称为派生类。在继承基类属性和行为的同时,派生类可以修改基类的某些行为,也可以添加自己的数据成员和函数成员。继承把派生类和基类联系了起来,派生类对象也可以被当作基类的对象使用。利用继承可以方便地重用已经定义的经过测试和调试的高质量的代码,提高软件开发的效率和软件质量。关于C++继承机制的介绍参见第14章。

4. 多态性

使用面向对象的方法进行程序设计时,系统中定义的很多类经常会基于继承的关系构成树(或图)形结构,多态性则为统一管理具有继承关系的不同类的对象提供了方便。使用多态性进行程序设计时,可以为具有继承关系的多个类定义统一的接口,而不同的类对于接口的实现则可各不相同。当通过统一的接口操纵这些对象时,程序可以根据被操纵对象的类型来确定具体该执行什么操作。这就像比赛前教练要求运动员进行练习时,只需要对所有人说一句"大家练习一下",不同的人听到这条指令后会做不同的事,长跑运动员会去跑步、乒乓球运动员会去练球……。教练勿须分别对不同的人下不同的指令,如对长跑运动员说"你去跑步",对乒乓球运动员说"你去练练球"等。而在结构化程序设计方法中,函数调用语句就确定了要执行的语句序列,如果教练要不同的运动员做不同的动作,就必须对不同的人下不同的命令。关于C++多态性的介绍参见第15章。

前面提到的Ada语言,虽然支持对象风格的程序设计,但不支持继承和多态性。这类语言称为基于对象的程序设计语言,而不是面向对象的程序设计语言。面向对象的程序设

计语言不仅要包含基于对象程序设计的特性,还应该支持继承和多态性。

毫无疑问,面向对象程序设计已经成为今天程序设计的主流。C++作为C的一个扩展,全面支持面向对象编程技术;Java作为最受欢迎的编程语言之一,也是一种面向对象的编程语言;Visual Basic发展成Visual Basic.NET后,也成为一种面向对象的编程语言。为什么现代程序设计语言如此倾向于面向对象编程呢?这是因为面向对象程序设计有很好的特点,如易于进行代码维护、程序可扩展性好、代码更容易重用等。这些特点都是结构化程序设计所欠缺的。面向对象方法为软件产品的扩展,以及质量保证中的许多问题提供了解决办法。这项技术能够大大提高生产力,并且可以提高软件的质量和降低软件维护费用。下面简单介绍面向对象技术的优点。

(1) 易于建模。允许将问题空间中的对象直接映射到程序中。以数据为中心的设计方法更容易抓住可实现模型的更多细节。

(2) 易于维护。面向对象程序设计的模块性是与生俱来的,其核心是类的设计。数据隐藏可以保护程序免受外部代码的侵袭,基于对象的工程可以很容易地分割为独立的部分。对象间通信所使用的消息传递技术使得对象与外部系统之间的接口描述更加简单,软件复杂度变得更加容易控制。

(3) 可扩展性好。面向对象编程支持扩展性。对象在程序中是一个个相对独立的包含数据和功能的实体,程序员可以向程序中增加一个新类或对象而不会影响到其他类的操作。面向对象的系统很容易从小到大逐步升级。

(4) 代码重用度高。继承可以大量减少多余的代码,并扩展现有代码的用途。如果已经有一个具有某种功能的类,则可以很快地扩展这个类,创建另一个具有更多功能的类。既然类将数据和功能封装到一个独立的实体中,以类为基础提供一个类库乃至整个应用程序框架就变得更加容易了。例如,现在经常使用的C++的一些编程环境(如Visual C++等)所提供的类库和应用程序框架,可以在标准的框架上构建程序,而不必一切从头开始,从而减少软件开发时间并提高生产效率。

以上关于面向对象程序设计的种种描述,还需要在学习面向对象的编程思想和进行面向对象程序设计的过程中进一步体会。

10.4 面向对象程序的结构

面向对象程序设计的主要任务是设计类,再以类为模板创建不同的对象,然后协调这些对象共同工作。和C一样,C++整个程序仍然需要一个入口,也就是main函数。下面是一个C++面向对象程序的实例。

【例10-1】 编写一个音像资料类Media,要求包含音像资料基本信息,能响应询问基本信息的消息,创建该类的对象,并通过消息机制输出对象的信息。

```
//ex10_1.cpp: 编写音像资料类 Media,创建该类的对象,并输出对象的信息
#include <string.h>
#include <iostream.h>
//定义媒体类型
#define ISAUDIO 1
```

```
#define ISVIDEO 2
//定义类 Media
class Media {
public:
    //构造函数
    Media(char * n)
    {   name=new char[strlen(n)+1];
        strcpy(name,n);
        type=0;
    }
    //析构函数
    ~Media() {delete [] name;}
    void setType(int t) {type=t;}
    void showInfo()
    {
        cout<<name<<"is a";
        if (type==ISAUDIO)
            cout<<"audio disc.";
        else if (type==ISVIDEO)
            cout<<"video disc.";
        else
            cout<<"unclassified disc.";
        cout<<endl;
    }
private:
    char * name;
    int type;
};
main()
{
    //声明类 Media 的对象 discDion 和 discHanks
    Media discDion("The Colour of My Love"), discHanks("Forrest Gump");
    discDion.setType(ISAUDIO);          //给对象 discDion 发消息 setType
    discHanks.setType(ISVIDEO);         //给对象 discHanks 发消息 setType
    discDion.showInfo();                //给对象 discDion 发消息 showInfo
    discHanks.showInfo();               //给对象 discHanks 发消息 showInfo
    return 0;
}
```

程序输出结果：

```
The Colour of My Love is a audio disc.
```

```
Forrest Gump is a video disc.
```

例10-1中的程序首先定义了类Media。类Media包含两个部分：一部分是关键字private后面的私有数据部分；另一部分是关键字public后面定义的4个公有成员函数。私有数据部分定义了一个字符指针name和一个int类型的数据成员type，类Media的所有对象自动拥有这些数据。4个成员函数中，第一个函数的函数名和类名Media完全相同，是构造函数，其作用是在创建对象时对对象中的数据进行初始化。第二个函数～Media是析构函数，其作用是在对象被撤销时做一些清理工作，在这里则是释放对象动态申请的字符数组空间，防止产生内存泄漏。第三个函数setType用来设置对象的私有数据type的值，type是类定义的私有数据，外界程序不能直接访问，只能通过定义在public：部分的公有成员函数访问。第四个函数showInfo则用于输出媒体资料的基本信息，语句

```
cout<<name<<" is a ";
```

先在屏幕上输出字符指针name指向的字符串，然后输出字符串"is a"，接下来的if-else嵌套语句则用来输出媒体的类型，最后再输出一个换行符(endl)。cout是输出流对象，它和标准输出流(屏幕)关联，cout和流插入运算符(<<)一起将要输出的数据显示到屏幕上。

在类Media之后程序定义了测试函数main，main函数首先使用声明：

```
Media discDion("The Colour of My Love"), discHanks("Forrest Gump");
```

声明了类Media的两个对象discDion和discHanks，并在声明对象的同时，通过隐式调用类Media的构造函数分别给对象discDion和discHanks中的name属性设置了初始值"The Colour of My Love"和"Forrest Gump"。然后分别给discDion、discHanks两个对象发消息(通过对象discDion或discHanks调用相应的成员函数)，输出两个对象的信息。

习　题　10

10.1　比较结构化程序设计和面向对象程序设计，为什么很多结构化程序设计语言都进行了面向对象的扩充？

10.2　对象之间都是通过相互发送消息进行交互，请思考：人(对象)和手机(另一个对象)之间发送哪些消息？

10.3　填空：

(1) 类的定义以________开始，以________结束。

(2) 在屏幕上输出信息的对象是________。

(3) 计算机能直接理解的语言是________。

(4) 面向对象语言具有________、________和________的特性。

10.4　执行下列C++语句将输出什么内容？假设x的值为5，y的值为10。

(1) cout<<(x=y);　　(2) cout<<(x+y);

(3) cout<<"x="<<y<<endl;　　(4) cout<<"x+=y";

(5) cout<<(x+=y)<<endl;　　(6) cout<<(char)(x+y);

10.5　下面哪些不是对象的特点：

(1) 多态性　　(2) 递归性

(3) 抽象性　　　　(4) 结构性

(5) 封装性

10.6 试用面向对象的方法分析一个图书馆管理系统,定义图书管理员类 Librarian,在该类的定义中应该包含哪些函数和数据?

10.7 把教室作为一个对象来考虑,那么相应的类如何定义?讨论与之相关的抽象、数据、函数、封装、接口、信息隐藏。

10.8 将公路上的车辆作为对象考虑,它可以包含哪些数据和函数?尝试定义这样的一个类。

第11章 类与对象

【学习内容】

本章介绍类与对象。主要内容包括：

- ◆ 数据抽象的概念。
- ◆ C++ 中类的特点和作用。
- ◆ 类的定义。
- ◆ 类的访问控制。
- ◆ 类的数据成员和成员函数。
- ◆ 类的静态成员。
- ◆ 类的对象。
- ◆ 类的构造函数，包括构造函数的定义、作用和执行时机。
- ◆ 类的析构函数，包括析构函数的作用和执行时机。

【学习目标】

- ◆ 了解数据抽象的概念。
- ◆ 初步了解如何使用面向对象的方法分析系统。
- ◆ 掌握类的定义方法。
- ◆ 掌握类的数据成员和成员函数的定义方法。
- ◆ 掌握类的静态成员的作用及使用方法。
- ◆ 掌握如何在类的定义中使用访问控制策略。
- ◆ 掌握构造函数的定义方法、作用和执行时机。
- ◆ 掌握析构函数的定义方法、作用和执行时机。

C++ 是在 ANSI C 基础上发展而来的，既然使用 ANSI C 几乎可以解决所有的编程问题，为什么还需要 C++ 呢？设计和实现一门新的编程语言无非是出于几种考虑：一是语言的能力，新的编程语言是否能提供新颖的解决问题的方法或思路，是否能解决其他语言解决不了的问题，或者对于同一个问题，使用新语言是否能比其他语言节省很多工作量；二是语言的运行效率，编写同一段程序，使用新语言编写是否能比用其他语言要快很多；三是软件质量问题，使用新语言编程是否比用其他语言更容易编写出高质量的程序；四是安全问题，新语言是否提供更好的安全机制等。从本章开始，将系统学习使用 C++ 进行面向对象程序设计的方法，通过学习可以体会到用 C++ 进行面向对象程序设计比结构化程序设计具有更

多的优势。

C++对ANSI C语言最大的扩充就是增加了面向对象的概念,即增加了类和对象的概念。本章将介绍类和对象的基本概念,包括如何使用面向对象的方法进行分析,如何进行类的定义和对象的创建,以及构造函数和析构函数的作用等。

11.1 数据抽象的概念

面向对象程序设计方法是围绕现实世界的概念来进行建模的程序设计方法,它采用对象来描述问题空间的实体。如何理解一个对象呢?可以从两个角度来考虑:一是从程序设计的角度,对象可以理解为对"数据及这些数据上的操作"的封装;二是从真实世界的角度,对象是问题空间的实体。例如,在图书馆管理系统中,涉及的对象实体有书库、图书馆管理员、读者等,当然一个图书馆可能涉及多个管理员和读者。

一个软件公司又包含哪些实体呢?首先公司本身就是一个实体,公司下属的各部门也都是实体,如人事部、财务部、软件开发部、售后服务部、技术支持部、市场部等;再往下,每个部门里的每个工作人员也都可以看做是实体,如程序员甲、程序员乙等。如果将现实世界和程序设计联系起来考虑,需要由程序处理的现实世界中的任何事物都可以视为一个对象。

根据上面的分析不难看出,面向对象的程序设计方法在系统中是采用对象去描述和组织这些实体。要用面向对象的方法设计和实现一个系统必须满足以下条件。

(1) 设计的系统必须能很好地描述问题空间的实体。如果要将"软件开发部"这个实体描述为一个对象,这个对象就必须包含软件开发部门所涉及的各种数据,如部门有多少人、人员之间的关系、当前开发的项目、每个项目的进度、每个项目所产生的文档、每个项目的负责人和参加人等。除了数据之外,该对象还必须能实现软件部门的一些工作,如项目的立项、项目的交付、人员的调入和调出、人员的分配、与同级各部门之间的各种交互、向上级机构汇报等。

(2) 对象之间必须能互操作或互通消息。如图书馆管理系统里的一个借书操作涉及读者、图书馆管理员和书库3个对象:由读者向图书管理员提出申请,图书馆管理员根据申请内容向书库查询相应的图书信息,书库返回该图书的库存信息后,图书管理员执行取书操作,将图书从书库取出交给读者。这个操作必须由3个对象协同工作才能完成。

(3) 对象之间必须允许存在某种关系(如包含关系等)。"软件公司"作为一个复杂对象可以包含诸部门等较小的对象,部门对象又可以包含人员这些更小的对象。具有包含关系的各对象之间也必须能实现互操作或互通消息。

上述是对"使用对象去描述问题空间中的实体"的基本要求,也就是对数据抽象的基本要求。什么是数据抽象呢?抽象就是提取和表现事物的核心特性,忽略与当前问题无关的那些细节,以便更充分地注意与当前问题有关的方面。抽象不是去了解和描述问题的全部,而只是关注其中相关的一部分。如前面的例子,描述一个读者时,不关心他(她)的身高、体重,而只关心他(她)的姓名和借书证号;也不关心他(她)的吃饭、睡觉、运动等行为,而只关心他(她)的注册、借书、还书等行为。因为只有这些特性和行为才是要实现的图书馆管理系统真正需要关心的。数据抽象则是指在对问题空间的实体进行抽象的基础上定义两部分内容:一部分定义描述该实体的数据;另一部分定义对这些数据的操作,并且限定只能通过这

些操作对对象的数据进行修改和观察。

数据抽象本身也是一个比较抽象的概念，数据抽象的过程一般可以分为以下 4 个步骤来完成。

① 确定问题空间，明确要解决的问题。例如，要实现一个图书馆管理系统，一般只需要实现图书管理、借书、还书、读者管理等基本功能。

② 确定问题空间的实体。例如，明确读者、管理员、书库等实体。

③ 对实体进行抽象。例如，对于读者这个实体，可能需要描述姓名和借书证号、借还书的历史记录等；还需要实现几个操作，如注册、借书、还书等。

④ 用数据结构和函数对上述分析结果进行描述。

掌握上述过程就可以开始用面向对象的方法来分析、描述和解决现实世界的实际问题了。

11.2 抽象数据类型

面向对象程序设计允许对问题空间的实体进行抽象，抽象的结果是对实体的属性（数据）和行为（函数）的封装。抽象的结果通常用抽象数据类型来描述。

一个抽象数据类型包括以下 3 方面内容。

(1) 数据对象的一个集合。

(2) 作用于这些数据对象的抽象运算（不依赖于具体实现）的集合。

(3) 这种类型对象的封装，即除了使用类型中所定义的运算外，用户不能对这些对象进行操作。

C++ 用类（class）来描述抽象数据类型。ANSI C 中的结构（struct）只能定义数据集合，不能包含操作。C++ 对结构进行了扩充，也可使用 struct 定义抽象数据类型，但结构和类有差别：结构中成员的默认属性是公有的，而类中成员的默认属性则是私有的。此外，在抽象数据类型的使用上，class 比 struct 更常用，因此本书的抽象数据类型只使用 class，而不用 struct。

实际上，C++ 对所有预定义的内部数据类型都定义了相关的属性和操作。例如，long 类型，编译器知道一个 long 类型的变量占有多少字节的空间，也知道 long 类型的数据可以和另外一个 long 类型的数据进行加、减、乘、除等算术运算以及比较大小等关系运算等。因此，也可以说是一种数据类型，包含了它相关的属性和操作。C++ 将抽象数据类型作为一种新的数据类型，那么这种新的数据类型也必须包含相关属性和操作，更进一步，还提供了封装机制。

如果定义了一个类 Stack 来实现栈的功能，根据数据抽象的概念，Stack 必须封装栈的属性和行为，也就是必须要能存储放入栈中的数据，并且能执行压栈、弹栈、栈空判断、栈满判断等操作。

C++ 中的类可以说是 C 中结构的延伸，本节将通过例子来比较、说明使用类可以弥补结构的不足。

11.2.1 封装与信息隐藏

为了提高软件开发的效率,一般都尽可能地使用系统或其他程序员提供的库。前面已经使用过一些系统的库函数,如头文件 stdio.h 中定义的标准输入输出库函数等。打开文件 stdio.h,不难发现 C++ 提供的库中定义的是一些函数的原型和结构。程序员可以在程序的任何地方用这些结构来声明自己的变量,也可以在任何地方任意修改这些变量的值,库中定义的函数也都可以随意调用。这里面就可能存在一些错误隐患,如库 time.h 中的描述时间的结构 tm,里面包含年、月、日、时、分、秒等属性。但是,如果声明了一个 tm 类型的变量后,甚至可以把其中的分钟属性修改为 120,显然这就造成了与客观世界不相符合的错误。

C++ 的封装和访问控制机制为这个问题提供了解决方案。封装是面向对象的重要特征之一,是对象和类概念的基本特性。类把数据和函数封装起来,并定义这些函数和数据的访问控制属性,类可以决定哪些成员(包括数据和函数)可以被外界直接访问,哪些成员只能被类自己的成员函数访问。一般情况下,不要直接访问一个对象的内部数据,而是将对象的内部数据定义为私有成员,然后通过操作接口(公有函数)来访问这些私有数据,这就是信息隐藏。封装和信息隐藏可以保证对数据的修改只来自于类的内部,外界对这些数据的修改只能通过有限的公有方法来实现。对于上面时间结构的问题,如果将年、月、日、时、分、秒等属性作为私有属性,外界对这些属性的访问和设置就只能通过专门的公有函数来完成,而在这些公有函数内将检查并保持这些属性满足相应的约束,就可以避免将属性修改为无意义的值的问题了。事实上,信息隐藏是用户对封装的认识,封装为信息隐藏提供支持。封装保证了对象具有较好的独立性,对程序的维护修改就更加容易。

程序设计方法提示: 理论上,类的所有的数据成员都应该作为私有成员,不能由外界直接操作,而只能通过公有函数来访问和设置这些私有成员。

11.2.2 接口与实现的分离

C++ 的类允许将接口和实现分离开来。类的作用之一就是创建该类的对象,然后通过调用对象的成员函数(给对象发送消息)完成一定的功能。类的使用者关心的是类的接口——类的公有数据成员和函数;类的设计者关心的是类的实现,这可以使程序员将精力都集中到自己该关心的部分。外界使用对象时,只能通过对象的公有成员函数进行,而不必关心对象的内部数据和实现细节,从而避免编写出依赖对象实现细节的代码。因此,当对象的代码修改以后,只要公有函数的接口(包括参数列表和返回值)保持不变,使用该对象的外部程序可以勿须修改就能够适应修改后的对象,这样可以将修改程序带来的影响减少到最低程度。

对于需要合作开发的软件项目,接口和实现的分离机制非常有用。如果需要给别人提供程序模块,但同时又不希望别人了解程序模块的实现细节,这时就可将模块的实现部分(执行程序)和模块的接口部分(类的定义)分开。接口部分被定义在一个头文件中,而实现部分则在一个或多个程序文件中被编译成可动态装配的库(dll 文件)。

一般情况下,使用别人提供的模块时,只关心这个模块的接口:如何使用这个模块?怎么调用其中的函数?它会返回怎样的结果?对函数的具体实现并不会关心,因为这毕竟应该是别人的工作。

如果只在 Microsoft 操作系统环境下使用自己的系统，动态链接库就是实现共享的一个很好的机制，C++ 接口与实现的分离机制也为动态链接库提供了极好的支持。例如，若需要使用 A 公司提供的一个模块完成系统开发，A 公司提供的将不会是模块的源程序，而是头文件及一个相关的动态链接库。头文件中描述了如何使用该模块，如外部函数的函数原型或类的定义。在使用该模块时可能会发现该模块存在缺陷，就会通知 A 公司进行修改。A 公司的修改一般也只限于模块的实现部分，因为接口部分是经过双方讨论后确认的，需要修改的可能性较小，所以通常修改的结果只是获得一个新的动态链接库而不是一个新的头文件。在这种情况下，使用库的程序不需要进行任何修改，甚至不需要重新编译就可以直接使用新的动态链接库进行测试运行。

11.2.3 用结构实现用户定义类型栈

C++ 语言编程的重点是类而不是函数。C++ 中的类可以看成是结构的自然延伸，在学习类和对象之前，先重温“结构”，并用结构去实现用户定义类型栈 Stack。栈是按先进后出的原则在内存中组织的一个存储区域。该区域一端固定一端活动，固定端称为栈底，活动端称为栈顶。往栈中存入或取出数据都在栈顶进行，新压入栈中的数据放在栈顶，从栈中取数据时也是获取栈顶的数据，因此，先压入栈的数据总是在后面被取出，栈的结构如图 11-1 所示。

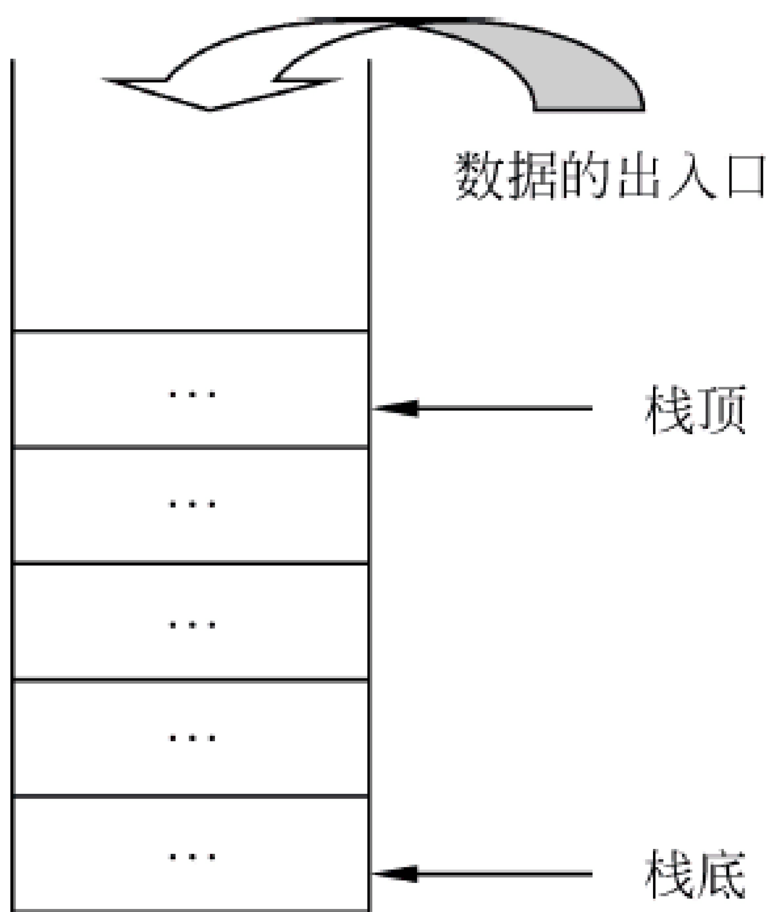


图 11-1　栈结构

例 11-1 中程序定义的结构 Stack 包含 3 个数据成员：data 成员用来存储栈元素，memNum 记录存入栈中的元素个数，size 记录初始化栈时动态分配的空间能存储的栈元素的个数。栈在功能上需要实现栈的初始化、压栈、弹栈、释放栈等功能，在数据上要实现栈空间的分配和栈数据的存储。

【例 11-1】 用结构实现栈，通过函数实现压栈、弹栈、栈的初始化和栈的释放等功能，并用两种不同的方式（通过栈函数和直接操作栈的内部数据）来使用栈。

```
//ex11_1.cpp: 用结构实现栈,并用两种不同的方式使用栈
#include <iostream.h>
struct Stack {                                  //结构定义
    int * data;                                 //栈数据存储
    int memNum;                                 //栈元素个数
    int size;                                   //栈大小
};
int initStack(Stack &s, int size);              //初始化栈的函数原型
void delStack(Stack &s);                        //释放栈的函数原型
int popStack(Stack &s, int &num);               //弹栈的函数原型
int pushStack(Stack &s, int mem);               //压栈的函数原型
//初始化栈
int initStack(Stack &s, int size)
{
```

```
        s.data=new int[size];
        if (s.data==NULL)
            return 0;
        s.memNum=0;
        s.size=size;
        return 1;
    }
    //释放栈
    void delStack(Stack &s)
    {delete [] s.data;}
    //弹栈,无数据时返回0,否则返回1
    int popStack(Stack &s, int &num)
    {
        if (s.memNum==0)
            return 0;
        num=s.data[--s.memNum];
        return 1;
    }
    //压栈,成功返回1,否则返回0
    int pushStack(Stack &s, int mem)
    {
        if (s.memNum==s.size)
            return 0;
        s.data[s.memNum++]=mem;
        return 1;
    }
    int main()
    {
        int i, num;
        Stack newStack;
        initStack(newStack, 10);
        //压栈
        cout << "Push integers to stack through function pushStack :" << endl;
        for (i=0; i<10; i++)
        {
            cout<< i << " ";
            pushStack(newStack, i);
        }
        cout << endl;
        //弹栈
        cout << "Reading from function popStack :" << endl;
        for (i=0; i<10; i++)
        {
```

```
        if (popStack(newStack, num))
            cout<< num << " ";
    }
    cout<< endl << endl;
    //直接操作栈的内部数据,破坏性地使用栈
    //直接将数据存入 data
    cout << "Push integers directly into the data in stack :" << endl;
    for (i=10; i<20; i++)
    {
        cout<< i << " ";
        newStack.data[newStack.memNum++]=i;
    }
    cout<< endl;
    //直接从 data 中读取数据
    cout << "Reading directly from the data in stack :" << endl;
    for (i=0; i<10; i++)
        cout<< newStack.data[i] << " ";
    cout << endl;
    //非法读取 data,造成越界
    for (i=10; i<20; i++)
        cout << newStack.data[i] << " ";
    cout << endl;
    //释放栈
    delStack(newStack);
    return 0;
}
```

运行结果:

```
Push integers to stack through function pushStack :
0 1 2 3 4 5 6 7 8 9
Reading from function popStack :
9 8 7 6 5 4 3 2 1 0
Push integers directly into the data in stack :
10 11 12 13 14 15 16 17 18 19
Reading directly from the data in stack :
10 11 12 13 14 15 16 17 18 19
-33686019 0 0 0 97 113 4397312 4397104 4350140 21
```

主程序 main 函数首先声明了一个结构变量 newStack。语句

```
initStack(newStack, 10);
```

对 newStack 进行初始化。initStack 是一个函数,采用传引用的方式为结构变量 newStack 中的指针 data 分配了 10 个整型数据的空间。然后,main 函数对栈进行测试,第一个循环语句将 0~9 的 10 个整数压入栈中,第二个循环将栈中的 10 个整数弹出并输出到屏幕。到此

为止,main 函数都使用程序提供的栈操作函数来操作栈。

使用结构实现栈时,由于变量 newStack 是 main 函数的一个局部变量,main 函数可以直接访问其中的数据。语句

```
for (i=10; i<20; i++)
{
    cout<<i <<" ";
    newStack.data[newStack.memNum++]=i;
}
```

直接将 10～19 的 10 个整数存入 newStack 的 data 数组中。然后,main 函数使用循环语句直接从栈中读取数据,由于最后一个循环在读取数据时下标越界,输出的结果也不知所云。可见,结构并没有提供保护数据不被非法修改的机制。

对于函数调用,如果传递的是大型参数,如结构变量、对象等,为了避免复制参数的开销,一般采用传引用的参数传递方式或使用指针传递结构变量或对象的地址。

例 11-1 的程序在实现压栈和弹栈操作时,函数的返回类型都是 int,因为压栈和弹栈操作有可能不成功(栈空时弹栈会失败,栈满时压栈也会失败),因此函数需要通过返回值告知操作的结果。对于弹栈,由于返回值标志操作是否成功,弹出的栈元素通过函数的参数返回,所以 popStack 函数的第二个参数采用传引用方式获取栈元素的值。

结构存在一些明显的缺陷。对结构的初始化可以通过专门的函数实现,也可以在程序的任何地方直接进行。直接修改结构成员的值时,可能会对结构成员赋予不恰当的值。而且在程序执行的过程中,结构成员都有可能被设置成错误的值,因为程序可以直接操纵结构的成员,就像例 11-1,main 函数中直接对栈中的数据进行赋值,这种赋值不仅会破坏栈“先进后出”的原则,甚至会破坏整个栈中的内容。另外,因为所有程序都有可能是直接操纵结构,使得外界使用该结构的程序和结构的内部细节紧密相关。如果结构的定义被修改,所有使用该结构的程序都有可能需要修改。出现这种情况的原因在于结构没有隐藏其内部数据的细节,没有将实现细节和对外接口分离。如果能够隐藏结构的组成,做到实现细节和对外接口的分离,并为外部使用者提供统一的接口,则当结构发生变化时,程序员只需修改接口函数的内部实现而保持接口不变,而且所有使用结构的程序都不必修改了。

结构还有另外一些问题。作为一种数据类型,都有一些可以直接作用于该类型数据的运算符。例如,+可作用于 int 类型的数据上,int 类型的数据也可直接输出。结构就不行了,除了=能实现两个结构变量之间的赋值以及 & 能获取结构变量的地址外,其他运算符都不能适用于结构。如果要增加对结构的操作,就必须通过设计特定的函数来实现,如上面针对 Stack 结构的 popStack、pushStack 等函数。显然这些函数与 Stack 紧密相关,但是程序中并没有体现出这种相关性。

C++ 中的类解决了上述问题。首先,类对数据和函数进行了封装,显式地给出了数据和相关操作的相关性;其次,类对外还提供了接口——公有成员函数。对对象的数据成员的初始化和修改都只能通过特定的成员函数进行,从而保证了数据的一致性;由于类提供了外部访问的接口,只要保证接口不变,类的实现细节的改变不会影响其他程序,即使出错也能把错误局限于类中,不会延伸到其他程序中。这样,显著增强了系统的可维护性。

第 13 章将介绍运算符重载的概念,通过运算符重载,可以把运算符直接作用到对象上,

并能通过流插入运算符(<<)和流提取运算符(>>)完成整个对象的输入输出。

下节用类重新实现栈,以说明用类建立抽象数据类型带来的好处。

程序设计方法提示:定义函数时,如果要传递的参数是大型参数,如结构变量、对象等,为了避免复制参数的开销,一般采用传引用的参数传递方式或者使用指针传递结构变量或对象的地址。

11.2.4 用类实现抽象数据类型栈

类可以封装数据和函数,用类实现栈就是定义这样一个类,它封装了与栈相关的数据和函数,并提供了操作栈的接口。C++ 中的抽象数据类型通常使用关键字 class 来定义。类的成员函数有时也叫"方法",成员函数的作用就是响应发送给对象的消息。类的成员数据和成员函数统称为类的成员。

通过定义类可以建立一个抽象数据类型,使用类名可以声明该类型的变量,也就是对象。例 11-2 中的程序定义了类 Stack,实现了抽象数据类型——栈。

【例 11-2】 类 Stack 的定义。

```
//文件 stack.h: 类 Stack 的定义
class Stack {
public:
    Stack(int s);                       //构造函数
    ~Stack();                           //析构函数
    int pop(int &num);                  //弹栈
    int push(int num);                  //压栈
private:
    int * data;                         //栈数据存储
    int memNum;                         //栈元素个数
    int size;                           //栈大小
};
```

类的定义以关键字 class 开始,Stack 是类的名字,也就是定义的抽象数据类型的名字。花括号括起来的部分是类的体,类的定义以分号结束(在右花括号后面)。

在类的定义中,public 和 private 是类成员的访问说明符,说明了类成员的访问属性。放在 public 之后并在下一个访问说明符之前的类成员具有公有属性,程序中任何能访问到类的对象的地方都能直接访问到该对象的公有成员(包括公有成员函数和公有数据成员)。放在 private 之后并在下一个访问说明符之前的类成员具有私有属性,私有成员只能被该类的成员函数访问到。每一种类成员的访问说明符都可以在类中多次出现。

结构只能封装数据,而且结构里的成员可以被程序直接操作。与 ANSI C 相比,C++ 对结构进行了扩充,struct 和 class 一样,不仅可以封装数据和函数,而且可以设置成员的访问属性,用 struct 可以定义和类一样的抽象数据类型。唯一不同的地方就是对于不属于任何访问说明符的成员,在 struct 的封装下是公有的,而在 class 的封装下是私有的。在建立抽象数据类型时,class 比 struct 更常用。

类 Stack 中访问说明符 public 后面有 4 个函数(Stack、~Stack、pop 和 push)的函数原型,它们是类的公有成员函数,组成了类的接口。对于类的使用者来说,一般只关心其公有

成员函数,因为它们也只能访问类的公有成员。

4个函数中有两个特殊的函数,和类名完全相同且没有返回类型的是构造函数。构造函数一般在创建类的对象时用来初始化对象的数据成员,在一个类的对象被创建时(不管是声明对象或用 new 操作符动态创建对象),类的构造函数会被自动调用,而且只在创建对象时调用一次。构造函数可以确保对象在使用前被初始化。

另外一个与类名相似(在类名的前面多了一个字符~)、也没有返回类型的函数是析构函数。析构函数一般用来完成撤销对象时应该做的工作,如释放类的对象在生存期间动态申请的空间。C++动态申请的空间只能在程序中动态释放,系统无法自动回收。为了防止空间的浪费,必须及时释放程序中动态申请的空间。在一个类的对象被撤销时,类的析构函数会被自动调用,因此,析构函数是释放对象动态申请空间的最佳时机。

类的 private 部分包含3个数据成员,它们只能被类的成员函数(还有下一章将要介绍的类的友元)访问。类的数据成员一般都放在 private 部分,但也可以放在 public 部分(不提倡),供外界访问的类的成员函数一般都放在 public 部分,其他只是为另外一些成员函数提供支持的成员函数,则放在 private 部分。

这里将类的3个数据成员放在 private 部分,保证了只有类的成员函数才能操作它们,从而确保了只要类被正确实现,类的对象的数据就不会被外界破坏。

定义好类之后,类名就可以作为新的数据类型来使用,就像一些内部类型一样,程序员可以用类名来声明变量(对象)、数组、指针等。例如:

```
Stack oneStack;                         //Stack类型的对象
Stack arrayOfStack[10];                 //Stack类型的对象数组
Stack * pStack;                         //Stack类型的指针
Stack &s= oneStack;                     //引用一个 Stack对象
```

上面讨论的是类的定义,注意类 Stack 的定义在一个头文件里(文件名为 stack.h)。类的实现部分一般和类的定义分开,放在另一个文件(stack.cpp)中。

【例 11-3】 类 Stack 的实现。

```
//文件 stack.cpp: 类 Stack的实现
#include "stack.h"
#define DEFAULTSIZE 100
//构造函数
Stack::Stack(int s)
{
    size= s> 0?s:DEFAULTSIZE;
    data= new int[size];
    memNum= 0;
}
//析构函数
Stack:: ~Stack()
{delete [] data;}
//弹栈函数,不成功返回 0,成功则返回 1,栈元素由参数返回
```

```
int Stack:: pop(int &num)
{
    if (memNum==0)
        return 0;
    num=data[--memNum];
    return 1;
}
//压栈函数,不成功返回 0,成功则返回 1
int Stack:: push(int mem)
{
    if (memNum==size)
        return 0;
    data[memNum++]=mem;
    return 1;
}
```

类的实现也可以和类的定义放在一个文件里,但一般不提倡这样做,把类的定义和类的实现分开会让程序的结构显得更加清晰。

文件 stack.cpp 中的第一条语句:

```
#include "stack.h"
```

包含了类 Stack 的定义文件,这是必须加的,因为该文件后面的程序会用到类 Stack 的定义信息。

类 Stack 的构造函数有一个参数,指定了栈空间的大小。构造函数对类的 3 个私有数据成员进行了初始化,为 data 分配了适当大小的空间,在 size 中记录栈空间的大小,将 memNum 初始化为 0。仔细观察构造函数的实现,它没有返回类型,在函数名 Stack 前面多了一个标志 Stack::,其中 Stack 是类名,::是二目作用域运算符,表示该函数是类 Stack 的成员函数。类的构造函数可以确保类的所有对象在使用前都被正确地初始化。

析构函数释放了构造函数动态分配的栈空间,对于对象生存期间动态分配的空间,析构函数是最好的释放时机。

另外,两个成员函数(压栈和弹栈)的实现与上一节用结构实现时的函数基本相同,只是在函数名前面加上了类名和作用域运算符,说明该函数是成员函数,属于类 Stack。和上一节相比,两个函数都少一个参数,因为作为类的成员函数,它知道该把数据放到什么地方,也知道从哪儿获得数据(即从接收消息的对象的数据成员处获得数据,这在第 12 章 this 指针时详细描述)。

成员函数的实现一般和类的定义分开,并放在不同的文件里,这时在函数名前面必须加上类名和作用域运算符以声明该函数属于某个类。成员函数的实现也可以放在类的定义里面,放在类里面的成员函数实现不需要加类名和作用域运算符。把函数的实现放在类里会使类的定义变长,从而使结构不清晰。

类的数据成员可以在类中声明,但不能在声明时初始化,对数据成员的初始化只能在构造函数中进行,或通过类的成员函数进行修改。常见的程序设计错误是在类中声明数据成员时同时对它(们)初始化。

到这里,栈类 Stack 的定义和实现都已经完成,下面编写一个程序去使用它。类的用户一般只关心类的接口部分,也就是这个类能完成哪些功能。对于栈 Stack,它的接口部分包含 4 个成员函数,但构造函数和析构函数都是自动调用的,能使用的就是压栈和弹栈两个操作。

例 11-4 中的程序使用了类 Stack,并对其中的初始化、压栈、弹栈操作进行了测试。

【例 11-4】 创建例 11-2 中定义的类 Stack 的对象,然后压入和弹出超过栈空间大小个数的数据,注意类 Stack 对内部数据的保护方法。

```
//文件 ex11_4.cpp: 测试栈类 Stack
#include "stack.h"
#include <iostream.h>

int main()
{
    int i, num;
    Stack myStack(10);

    //压栈
    cout<<"Push integers onto stack : ";
    for (i=0; i<11; i++)
    {
        if (!myStack.push(i))
            cout<<endl<<"Push "<<i<<" failed."<<endl;
        else
            cout<<i<<" ";
    }

    //弹栈
    cout  <<"Pop integers from stack : ";
    for (i=0; i<11; i++)
    {
        if (myStack.pop(num))
            cout<<num<<" ";
        else
            cout<<endl<<"Pop failed.";
    }
    cout<<endl;

    return 0;
}
```

运行结果:

```
Push integers onto stack : 0 1 2 3 4 5 6 7 8 9
Push 10 failed.
Pop integers from stack : 9 8 7 6 5 4 3 2 1 0
Pop failed.
```

例 11-4 中的程序使用了类 Stack。使用一个类时可以只关注类的定义,而不关心类的

实现,因此在文件 ex11_4.cpp 中需要包含头文件 stack.h。

例 11-4 中 main 函数先声明了类 Stack 的对象 myStack:

```
Stack myStack(10);
```

类 Stack 的构造函数需要一个整型参数以确定栈空间的大小,在声明对象时必须提供一个整数作为参数。对象声明时会自动调用类的构造函数对对象中的数据成员进行初始化,并将对象(myStack)后面圆括号中的参数传递给构造函数。如果类的构造函数没有参数,在声明类的对象时就不需要提供参数,对象后面的括号也不需要(和声明内部类型,如 int 类型的变量的格式相同)。

在 C++ 中,访问对象的成员的方式和访问结构变量的成员的方式相同,可以使用对象名或对象的引用加点操作符去访问,也可以使用类指针加箭头操作符去访问。例如:

```
Stack astack(10), * aptr= &astack;
astack.push(10);
aptr->push(10);
```

main 函数在声明对象 myStack 之后,对它进行操作。首先使用 for 语句循环 11 次,试图将整数 0～10 共 11 个数压栈,然后再试图使用 for 语句循环 11 次执行弹栈操作,并作判断,如果弹栈成功则打印栈元素。程序的运行显示其输出结果是 9～0 共 10 个数。由于栈只有 10 个空间,11 次压栈操作只有前 10 次成功,11 次弹栈操作也只有前 10 次成功,类的成员函数对私有数据成员进行了有效的保护。

比较本节用类实现的栈和上一节用结构实现的栈,不难发现,使用结构实现的栈是不安全的,因为程序中任何能访问到该结构变量的地方都能直接修改结构中的数据成员,从而破坏了栈的性质和内容,并且对栈结构的操作分散在程序的各个部分,如果要修改栈结构的定义,将会涉及程序的各个地方,程序的维护很不方便。

对于用类实现的栈来说,由于采用了封装以及接口和实现分离的技术,栈变得很安全,数据要入栈只能通过类的成员函数 push 完成,push 函数对数据空间大小进行了判断,保证操作不会越界。栈中的数据也只能通过成员函数 pop 取出,这两个函数共同保证了栈内数据的正确性。接口和实现的分离使得类的使用者只需要关心类的定义(或只关心类定义中的公有成员函数),类的实现被独立出来。如果需要修改类的实现,如栈的元素不用数组而改用链表来存放,则只需要修改 push 和 pop 函数的内部实现,保证这些公有函数的接口不变,类的使用者(main 函数)就可以勿须作任何修改,也能够使用修改后的 Stack 类,而不必关心 Stack 类是如何修改的。接口和实现的分离提高了程序的可维护性。

另外,对于用结构实现的栈,使用前必须记住要调用函数对栈进行初始化,而使用类实现的栈则根本不需要关心初始化问题,因为声明类的对象时,会自动调用类的构造函数完成对对象的初始化。

11.3 类和对象的定义

上一节介绍了如何定义类和声明类的对象,并演示了类的使用方法。本节将详细介绍与类和对象的定义相关的问题。

11.3.1 数据成员

可以在类中定义各种数据成员,包括各种类型的变量、指针、数组等,甚至可以定义另外一个类的对象(类的复合,将在第12章讲述)。定义类时只可以在类中声明数据成员,而不能对数据成员进行初始化,例11-5中的程序演示了给类中数据成员进行初始化时编译程序所报的错。

【例11-5】 定义类时对数据成员初始化出现的错误。

```
//文件 stack.h: 类 Stack的定义
class Stack {
public:
    Stack(int s);                          //构造函数
    ~Stack();                              //析构函数
    int pop(int &num);                     //弹栈
    int push(int num);                     //压栈
private:
    int * data;                            //栈数据存储
    int memNum=0;                          //栈元素个数
    int size;                              // 栈大小
};
```

编译报错:

```
error C2252: 'memNum' : pure specifier can only be specified for functions
```

上述程序编译时系统报错,错误信息显示对于类中的数据成员不能初始化(=0 被理解为声明成员函数是纯虚函数,参见第15章)。

类中的数据成员一般都放在private部分作为私有数据成员,然后使用成员函数去操作它们,这样做可以使数据的管理局部化,外部函数只能通过类的公有成员函数才能访问到对象的私有数据成员,将来即使数据成员发生变化,也只需要修改操作它们的成员函数的内部实现,而不会影响使用该类的外部程序代码。数据成员也可以放在public部分,但不是一种好的程序设计风格,如例11-6中的例子。

【例11-6】 不提倡的类的定义方式示例。

```
//文件 stack.h: 类 Stack的定义
class Stack {
public:
    Stack(int s);                          //构造函数
    ~Stack();                              //析构函数
    int pop(int &num);                     //弹栈
    int push(int num);                     //压栈
    int * data;                            //栈数据存储
    int memNum;                            //栈元素个数
    int size;                              //栈大小
};
```

上述程序中类的定义把所有的数据成员和成员函数都放在公有部分，这样的定义方式是合法的，但违反了信息隐藏的原则，存在隐患，因为声明栈对象之后，外部程序代码可以随意地直接修改对象的内部数据，有可能会破坏栈的约束和内容。

11.3.2 成员函数

类的成员函数的作用是响应发送给类的对象的消息，供外界使用的类的成员函数一般放在 public 部分，而由类自己使用的成员函数一般作为私有成员。类的公有成员函数构成了类的接口。从前面的例子可以看出，类的成员函数的实现一般放在类的外面，最好与类的定义不在同一个文件中，这种写法的好处是支持接口与实现的分离，使程序的结构更加清晰。成员函数的实现也可以在类的里面。

【例 11-7】 在类里实现成员函数。

```
//文件"stack.h": 类 Stack的定义
class Stack {
public:
    Stack(int size);
    ~Stack();
    int pop(int &num)                    //函数定义
    {   if (memNum==0)
          return 0;
          num=data [--memNum];
          return 1;
    }
    int push(int mem);                   //函数原型
    int * data;                          //栈数据存储
    int memNum;                          //栈元素个数
    int size;                            //栈大小
};
```

上述程序中类 Stack 的成员函数 pop 的实现部分直接写在类里，这时不需要函数原型，直接写出函数实现即可，函数名前面也不需加类名和作用域运算符(::)。编译器将这类成员函数按内联方式处理。一般情况下，除了极短的函数外，成员函数的实现都写在类的外面，否则会影响程序的可读性和目标代码的大小。

有些类的成员函数，如果它们的使用者只有类的其他成员函数，就可以放在 private 部分，这些成员函数也称为工具函数，它们只能被类的其他成员函数调用。

程序设计方法提示：除极短的函数外，建议类的所有成员函数的实现都放在类的外面。

下面简单介绍类的成员的作用域问题。C++ 规定类的成员(包括数据成员和成员函数)具有类作用域。具体来说，在类中定义的成员名可在类的范围内(包括类的定义和实现)或类的派生类中使用，或者通过对象名或对象的引用加点操作符(.)、类指针加箭头操作符(->)、类名加作用域运算符(::)等方式进行访问。

11.3.3 访问控制

前面介绍了两个访问说明符 public 和 private。C++还有另外一个访问说明符 protected(受保护的访问方式,将在第14章介绍),它们共同控制对类的成员函数和数据成员的访问。在 class 定义的类中,默认的访问方式是 private,所以类中第一个访问说明符之前的所有成员都是私有的。在一个类中,public、private 和 protected 3个访问说明符都可以多次使用,但最好不要这样用,因为会影响程序的可读性。

程序设计风格提示: *在类的定义中,public、private 和 protected 声明的成员要相对集中,并将 public 声明的成员放在前面,然后是 protected 声明的成员,private 声明的成员放在最后。*

类的私有成员只能被类的成员函数(或友元)访问,外部函数试图访问类的私有成员将导致语法错误。成员函数访问私有属性的例子已在例11-3中看到,类 Stack 的成员函数 pop、push 以及构造函数、析构函数都访问了该类的私有成员。下面是外部函数试图访问类的私有成员的例子。下面例子中的程序使用了类 Stack,但它试图访问类 Stack 的对象 myStack 中的私有成员。

【例 11-8】 编写程序使用例11-2中定义的栈类 Stack,并试图直接访问其实例化对象的私有数据成员,记录编译信息。

```
//文件 ex11_8.cpp: 使用栈 Stack
#include "stack.h"
#include <iostream.h>

int main()
{
    int i, num;
    Stack myStack(10);

    myStack.size=11;                //错误语句
    myStack.data[0]=0;              //错误语句

    return 0;
}
```

编译结果:

```
ex11_8.cpp(9): error C2248: 'size': cannot access private member declared in class 'Stack'
ex11_8.cpp(10): error C2248: 'data': cannot access private member declared in class 'Stack'
```

从例11-8中程序的编译结果可以看出,试图访问类 Stack 中的私有属性 size 和 data 都是不允许的。

类的公有成员可以被类的成员函数、友元(将在第12章中介绍)以及所有能访问到类的对象的外部程序代码直接访问,它们是类的对外接口。类的公有成员函数响应外部消息,也是外部函数访问类的私有属性的窗口。类的使用者通过类的接口知道类提供了哪些功能,而不必知道类的具体实现。

在设计类时最好把类的数据成员都放在 private 部分,如果外部函数有访问类的对象的

数据成员的需要,可以为外部函数提供设置或读取类的私有数据成员的公有成员函数,并在这些公有成员函数中保证数据成员的完整性和正确性。这种做法可以防止类的对象的数据成员被意外修改,从而提高程序的可维护性和可修改性。

类的访问说明符可以帮助实现信息隐藏和最低访问权原则。程序的最低访问权原则是指:针对程序中的数据,每个函数或模块都拥有最小的但却足够的访问权限。一般来说,最低访问权原则可以归纳为以下几点。

(1) 如果某函数不需要访问某些数据,则不给它访问的权限。如外部函数与类的私有属性的关系。

(2) 如果某函数需要读取某数据的值,但不需要修改它,则只给它读的权限,而不赋予它修改的权限。如将某些函数的参数设置为 const 属性。

(3) 只有当函数需要修改某数据时才给它全部的访问权限。

(4) 对于类的数据成员,即使外部函数需要修改,也不提倡给它直接的访问权限,而应该只给它提供公有成员函数接口,让它通过类的公有成员函数实现对私有数据的访问。

总之,把类的数据成员设置为私有的,这样能直接操作数据的就只有类的成员函数和友元,这可以减少程序出错的可能性,并易于维护。

11.3.4 静态成员

类是一种抽象数据类型,可以看成是一种模板。创建类的对象时,系统会为每一个对象分配独立的空间。对象只拥有数据成员的空间,类的成员函数被所有对象共享。在面向对象程序设计中,对象之间是相互独立的,消息是对象之间进行交互的唯一方式,一个对象无法获知另外一个对象的创建和撤销信息,这显然不方便对一个类的所有对象进行统一管理。有时,也需要对象之间能够通过共享数据空间来实现数据的交换。

程序如何实现空间的共享呢?一种解决方法是定义全局变量,这破坏了面向对象程序的封装特性,显然不可取;另一种方法是使用静态数据成员,类的静态数据成员可被该类的所有对象共享。类的静态成员包括静态数据成员和静态成员函数。静态数据成员提供了类的所有对象共享信息的一种方式。对于类的静态数据成员,系统只保留一份拷贝,该类的所有对象共享该拷贝,因此使用静态数据成员不仅能节省空间,还能使同一个类的不同实例对象保持对该数据的一致性。静态成员的声明以关键字 static 开始。

类的静态成员也可以是私有的、公有的或受保护的。静态数据成员属于类,被所有对象共享,即使在对象还不存在的时候(定义类之后,还没有创建类的对象之前),类的静态数据成员就已经存在,并可以访问它们。类的每个对象都有非静态数据成员的一份拷贝,访问非静态数据成员(属于类的某个对象)可以通过 3 种方式进行:对象名加点操作符、对象的引用加点操作符、类指针加箭头操作符。每种访问方式都必须有对象存在,因为非静态数据成员是属于每个对象的。静态数据成员则不同,除了可以通过对象访问它们外,还可以通过类名加作用域运算符的方式直接访问。

静态数据成员必须在定义类的时候就初始化,但不能在类里面初始化,只能在类外面进行。

静态成员函数是专门操作静态数据成员的成员函数,静态成员函数属于类,即使在没有对象存在的情况下,也可以通过类名和作用域运算符调用它,而非静态成员函数只能通过对

象调用(使用前面介绍的3种方法)。静态成员函数只能访问类的静态成员,不能调用非静态成员函数和访问非静态数据成员。注意,非静态成员函数可以访问静态数据成员,也可以调用静态成员函数。

下面例子改写了类Stack,希望能让类Stack知道它在某一时刻到底有多少对象实例存在。在看程序之前,不妨先思考,利用前面所学的知识有没有可能实现这个功能?如果有可能,和例11-9和例11-10中的方法进行比较,分析静态成员的优点在什么地方。

【例11-9】 定义能记录对象数目的Stack类(一)。

```
//文件 stack.h: 类 Stack的定义
class Stack {
public:
    Stack(int s);                      //构造函数
    ~Stack();                          //析构函数
    int pop(int &num);                 //弹栈
    int push(int num);                 //压栈
    static int getObjNum();            //静态成员函数,返回静态数据成员 objNum
private:
    int * data;                        //栈数据存储
    int memNum;                        //栈元素个数
    int size;                          //栈大小
    static int objNum;                 //静态数据成员,记录对象个数
};
```

为了记录类的对象的数目,需要有一个变量能被类的所有对象访问到。由于对象在产生时会自动调用类的构造函数,在撤销前会自动调用析构函数,所以要记录对象的数目只需在构造函数执行的时候将该变量加1,在析构函数执行时将该变量减1就可以了。例11-9中的程序选择类的静态数据成员记录对象的个数。

例11-9是类Stack的定义(文件stack.h),程序在类Stack中增加了两个静态成员:静态数据成员objNum和静态成员函数getObjNum,objNum用来记录对象数目,函数getObjNum则作为公有成员函数来供外界获取objNum的值。例11-10是类Stack的实现(文件stack.cpp)。

【例11-10】 Stack类的实现(二)。

```
//文件 stack.cpp: 类 Stack的实现
#include "stack.h"
//构造函数
Stack::Stack(int s)
{
    data=new int[s];
    size=s;
    memNum=0;
    objNum++;                          //一个对象产生时,objNum的值加1
}
```

```
//析构函数
Stack::~Stack()
{
    delete [] data;
    objNum--;                           //一个对象被撤销时,objNum的值减 1
}
//弹栈函数,不成功返回 0,成功则返回 1,栈元素由参数返回
int Stack::pop(int &num)
{
    if (memNum==0)
        return 0;
    num=data[--memNum];
    return 1;
}
//压栈函数,不成功返回 0,成功则返回 1
int Stack::push(int mem)
{
    if (memNum==size)
        return 0;
    data[memNum++]=mem;
    return 1;
}
//初始化类的静态数据成员
int Stack::objNum=0;
//实现类的静态成员函数
int Stack::getObjNum()
{
    return objNum;
}
```

因为类的静态数据成员不依赖于对象而存在,所以静态数据成员不能在构造函数中初始化,而只能在实现类时进行初始化。下面的语句将 objNum 初始化为 0:

```
int Stack::objNum=0;
```

和例 11-3 相比较,例 11-10 在构造函数中增加了一条语句,将 objNum 加 1,在析构函数中也增加了一条语句,将 objNum 减 1。同时,实现了静态成员函数 getObjNum,返回 objNum 的值。

声明静态成员函数或静态数据成员时需要以关键字 static 开始,初始化静态数据成员和实现静态成员函数时则不需要加 static。

修改完类 Stack 后,下面编写程序测试类 Stack,注意调用类的静态成员函数的方法。

【例 11-11】 在不同的地方以不同的方法创建例 11-9 和例 11-10 中定义的类 Stack 的实例化对象,并实时获取类 Stack 的对象个数。

```
//文件 ex11_11.cpp: 使用栈 Stack,并实时获取 Stack 的实例化对象的个数
#include "stack.h"
#include <iostream.h>

int main()
{
    cout<<"There is "<<Stack::getObjNum()<<" stack. "<<endl;

    {   //块 1
        Stack one(10);
        Stack * ptr=&one;

        cout<<"After stack one created, there is "
            <<one.getObjNum()<<" stack. "<<endl;

        {   //块 2
            Stack two(10);

            cout<<"After stack two created, there are "
                <<two.getObjNum()<<" stacks. "<<endl;
        }

        cout<<"After stack two destroyed, there is "
            <<ptr->getObjNum()<<" stack. "<<endl;
    }

    cout<<"After stack one destroyed, there is "
        <<Stack::getObjNum()<<" stacks. "<<endl;

    return 0;
}
```

程序运行结果：

```
There is 0 stack.
After stack one created, there is 1 stack.
After stack two created, there are 2 stacks.
After stack two destroyed, there is 1 stack.
After stack one destroyed, there is 0 stacks.
```

程序首先通过类名和作用域运算符(::)调用类的静态成员函数 getObjNum 获取当前对象个数，即语句：

```
Stack::getObjNum();
```

此时没有声明任何对象，所以对象个数为 0。程序进入块 1 后，声明类 Stack 的对象 one，自动调用构造函数，objNum 加 1，然后通过对象名加点操作符的方式调用函数 getObjNum，即语句：

```
one.getObjNum();
```

程序退出块 2 后，通过指向对象 one 的指针 ptr 调用函数 getObjNum，即语句

```
ptr->getObjNum();
```

运行结果显示，类 Stack 的定义获得了预期的效果。

11.3.5 对象的建立

定义好类之后，下一步就是声明类的对象。实际上，在前面的例子中，已经看到了对象的创建和撤销。类是一种抽象数据类型，像内部类型一样，可以声明类的变量（对象）、数组、指针、引用，也可以动态创建类的对象或对象数组。

分析下面例子中类 Circle 的定义。

【例 11-12】 圆型类 Circle 的定义。

```
//文件 circle.h：定义类 Circle

#if !defined __CIRCLE__H__
#define __CIRCLE__H__

#include <iostream.h>

class Circle {
public:
    Circle() {x=y=r=5.0;}

    void draw()
    {
        cout<<"Draw a circle at ("  <<x<<","
            <<y<<"), with radius "  <<r<<"."<<endl;

    }
private:
    double  x, y;                //圆心坐标
    double  r;                   //半径

};
#endif
```

上面程序定义了一个圆形类 Circle，它的 public 部分只包含一个构造函数和一个简单的 draw 成员函数。下面的语句声明了类 Circle 的对象、对象数组、指针和引用。

```
Circle one;                 //声明类 Circle 的对象 one
Circle array [10];          //声明类 Circle 的数组 array，包含 10 个数组元素
Circle * ptr=&one;          //声明类 Circle 的指针 ptr，并让 ptr 指向对象 one
Circle &o=one;              //声明类 Circle 的引用
```

也可以通过类指针动态创建类的对象，下面的语句动态创建了类 Circle 的对象和对象数组。

```
Circle * pone, * pten;      //声明 Circle 类型的两个指针
pone=new Circle;            //动态创建一个对象
pten=new Circle [10];       //动态创建对象数组
```

对象在创建时会自动调用类的构造函数，类 Circle 的构造函数没有参数，因此使用类 Circle 时就像使用内部类型一样。有些类的构造函数是需要参数的，如前面例子中类 Stack

的构造函数需要一个整型参数,在声明对象或动态创建对象时就需要提供参数,提供参数的格式和函数调用时提供参数的格式相同。例如,建立类 Stack 的对象:

```
//声明类 Stack 的对象 oneStack,并向构造函数传递参数 10(用圆括号)
Stack oneStack(10);

//声明 Stack 类型的指针 ptr,并让它指向一个动态生成的对象
//生成对象时给构造函数传递参数 20(用圆括号)
Stack * ptr= new Stack(20);
```

例 11-12 定义类 Circle 时使用了预处理指令 #define、#if !defined 和 #endif,使用这些预处理指令可以防止该头文件被一个源文件多次包含时导致编译错误(类 Circle 的重复定义错误)。其中,指令

```
#if !defined __CIRCLE__H__
```

判断标识符__CIRCLE__H__是否被定义,如果没有被定义则编译器编译后续程序,否则编译器跳过后续语句直到指令 #endif。指令

```
#define __CIRCLE__H__
```

则定义了标识符__CIRCLE__H__。

当某个源文件第一次包含头文件 circle.h 时,标识符__CIRCLE__H__没有被定义,编译器就会编译后续语句(包括类 Circle 的定义和标识符__CIRCLE__H__的定义),如果该文件第二次包含头文件 circle.h,这时由于标识符__CIRCLE__H__已经被定义,编译器就会跳过类 Circle 的定义,防止出现重复定义的错误。

程序设计方法提示: 由于类的定义可能会被很多头文件和源文件引用,预处理指令 #include 很可能会导致某个头文件在某个源文件中被包含多次。所以,建议在定义头文件时,通过使用上述预处理指令防止被多次包含时出现重复定义的错误。

11.4 构造函数

构造函数是一种特殊的成员函数,其名字和类名完全相同,没有返回类型,构造函数还有许多一般成员函数所没有的特性。本节将详细介绍构造函数。

11.4.1 构造函数的作用

对象的数据成员一般在构造函数中初始化,构造函数可以保证对象在创建后其数据成员都已经被正确初始化。构造函数是在对象建立时被自动调用的,不能指定其返回类型,试图指定构造函数的返回类型将导致语法错误。

【例 11-13】 编写在定义类 Circle 时试图指定构造函数的返回类型的程序,并观察编译结果。

```
//文件 ex11_13.cpp: 定义类 Circle
#include <iostream.h>

class Circle {
```

```
public:
    void Circle()                          //错误语句
    {
        x=y=r=5.0;
    }
    void draw()
    {
        cout  <<"Draw a circle at ("  <<x  <<","
              <<y  <<"), with radius "  <<r  <<"."  <<endl;
    }
private:
    double  x, y;                          //x、y坐标
    double  r;                             //半径
};
int main()
{
    Circle cobj;
    return 0;
}
```

编译结果：

```
error C2380: type(s) preceding 'Circle' (constructor with return type, or illegal redefinition of current class-name?)
```

上述程序在定义类 Circle 时试图指定构造函数的返回类型(void 是一种返回类型)，在编译时报错，错误信息指出在构造函数 Circle 前不能有类型名存在。

11.4.2 构造函数执行的时机

对象被声明或创建时，构造函数将被自动调用。一个程序往往会在不同的地方创建不同类的多个对象，这些类的构造函数总是随对象的产生而被调用执行。在程序的不同地方用不同的方式声明的对象具有不同的作用域，其构造函数执行的时机也不相同。

全局对象(定义在所有函数外面的对象，具有文件作用域)在所有程序执行之前就被创建。也就是说，在程序 main 函数第一条语句执行之前，全局对象已经被创建了，因此全局对象的构造函数在 main 函数开始执行之前被执行。

局部动态对象(定义在程序块内，具有块作用域)在程序执行到声明该对象的语句时被创建，因此局部动态对象的构造函数是在程序执行到声明该对象的语句时被执行。

局部静态对象(定义在程序块内，用 static 关键字声明，具有块作用域)在程序执行到声明该对象的语句时创建，因此同局部动态对象一样，局部静态对象的构造函数在程序第一次执行到声明该对象的语句时被执行。需要指出的是，局部静态对象只在它所在的程序块第一次执行时被创建一次，创建后即存在于程序的整个生命周期，所以当它所在的程序块再次执行时就不再被创建了。

动态创建的对象(用 new 运算符创建)在程序执行到该语句时被动态创建，因此动态创

建对象的构造函数在程序执行到该 new 操作时被执行。

11.5.2 节中的例 11-17 演示了全局对象、局部动态对象、局部静态对象和动态对象的构造函数和析构函数执行的时机。

11.4.3 构造函数重载

类的构造函数可以被重载以便根据不同的需要采用不同的方法初始化类的对象。重载构造函数的类具有多个构造函数,这些构造函数的函数名完全相同,都没有返回类型,而参数列表各不相同。和一般函数的重载一样,类的重载构造函数的参数列表(参数的个数、类型和顺序)必须不同,系统根据函数名和参数列表共同确定该调用哪个函数。类有多个构造函数时,根据声明对象或创建对象时提供的参数来确定调用哪个构造函数来初始化对象。

【例 11-14】 重载构造函数示例。

```
//文件 circle.h: 定义类 Circle
#if !defined __CIRCLE__H__
#define __CIRCLE__H__

#include <iostream.h>

class Circle {
public:
    Circle() {x=y=r=5.0;}

    Circle(double a, double b, double c)
    {
        x=a;
        y=b;
        if (c>0)
            r=c;
        else
            r=5.0;
    }

    void draw()
    {
        cout<<"Draw a circle at ("  <<x  <<","
            <<y  <<"), with radius "  <<r  <<"."  <<endl;
    }

private:
    double  x, y;                //x、y坐标
    double  r;                   //半径
};

#endif
```

类 Circle 有两个构造函数,一个不带参数,另一个带 3 个 double 类型的参数。这两个构造函数分别为:

```
Circle ();
```

```
Circle (double a, double b, double c);
```

它们在参数列表上有所区别。在声明对象时也可以有所选择，或者可以使用系统的默认值，x、y、r 的值都被赋值为 5.0，或者可以为它们设置希望的值。

```
//调用第一个构造函数,对象 obj 中的 x、y、r 都被赋值为 5
Circle obj;
//调用第二个构造函数,对象 obj 中的 x、y、r 分别被赋值为 4.0、5.0 和 6.0
Circle obj1(4.0, 5.0, 6.0);
```

注意：创建类的对象时会自动调用类的构造函数，创建一个对象只会根据参数列表调用类的一个构造函数，而且只调用一次，不会调用类的所有构造函数。

11.4.4 默认构造函数

在调用时不必提供参数的构造函数是默认构造函数。如果一个类有默认构造函数，则声明类的对象时可以不提供参数。下面例 11-15 中类 Circle 的第一个构造函数就是默认构造函数，所以声明对象时可以没有参数。例如：

```
Circle obj;                //调用默认构造函数
```

声明构造函数时，也可以为构造函数提供默认参数。给构造函数提供默认参数的好处是：即使在调用构造函数时没有提供参数，也会确保按照默认的参数对对象进行初始化。所有参数都是默认参数的构造函数也是默认构造函数。一个类不能同时拥有多个默认构造函数，因为编译器难以区分对它们的调用。

例 11-14 中的默认构造函数为对象的 3 个数据成员 x、y、r 都赋值为 5.0。第二个构造函数判断半径值是否满足条件，满足则直接赋值，不满足则赋默认值。如果使用默认参数，则可以将两个构造函数合并成一个构造函数。

【例 11-15】 使用默认参数的构造函数示例。

```
//文件 circle.h: 定义类 Circle
#if !defined __CIRCLE__H__
#define __CIRCLE__H__

#include <iostream.h>

class Circle {
public:
    //带默认参数的构造函数
    Circle(double a=5.0, double b=5.0, double c=5.0)
    {
        x=a;
        y=b;
        if (c>0)
            r=c;
        else
            r=5.0;
    }
```

```
        void draw()
        {
            cout<<" Draw a circle at ("<<x<<","
                <<y<<"), with radius "<<r<<"."<<endl;
        }
    private:
        double x, y;                          //x、y坐标
        double r;                             //半径
    };
    #endif
```

上面类 Circle 的构造函数使用了默认参数,因为所有参数都是默认参数,所以该构造函数也是默认构造函数。如果声明对象时提供了参数,则使用用户提供的参数值(相当于调用例 11-14 中的第二个构造函数);如果没有提供参数,则使用默认参数(相当于调用例 11-14 中的第一个构造函数,即默认构造函数)。

如果构造函数使用了默认参数,声明对象时不仅可以提供全部参数或不提供参数,还可以只提供部分参数。例如,声明类 Circle 的对象时可以不提供参数,也可以只提供 1 个、2 个或 3 个参数,对于没有提供的参数,构造函数用默认参数补足。可以使用下面的方法声明 Circle 的对象。

```
Circle  a;                              //相当于 Circle a(5.0, 5.0, 5.0);
Circle  b(1.0);                         //相当于 Circle b(1.0, 5.0, 5.0);
Circle  c(3.0, 4.0);                    //相当于 Circle c(3.0, 4.0, 5.0);
Circle  d(4.0, 5.0, 6.0);               //相当于 Circle d(4.0, 5.0, 6.0);
```

如果定义类时没有为类提供任何构造函数,系统会为该类自动提供一个参数列表和函数体都为空的默认构造函数。

11.4.5 复制构造函数

根据类的某个对象复制出一个完全相同的新的对象的构造函数称为复制构造函数。复制构造函数的参数应该是该类对象的引用。

下面的例 11-16 为例 11-15 中的类 Circle 添加了一个复制构造函数。

例 11-16 实现复制构造函数时,只简单地将对象参数 c 中的数据成员 x、y、r 的值复制给当前对象(使当前对象的各个数据成员和对象 c 中相应的数据成员的值完全相同)。

通过复制构造函数,程序可以创建和已有对象完全相同的新对象。例如:

```
Circle a(3.0, 4.0, 5.0);
Circle b(a);
```

第一个语句声明了对象 a,a 中数据成员 x、y、r 的值分别为 3.0、4.0 和 5.0(调用第一次构造函数);第二个语句声明对象 b,以 a 为参数调用复制构造函数,这样对象 b 中数据成员 x、y、r 的值和对象 a 完全相同,也分别为 3.0、4.0 和 5.0(调用第二个构造函数)。

如果类定义时没有定义复制构造函数,编译时会自动加入一个完成数据成员复制的复制构造函数。在函数调用或返回时,如果参数采取传值方式传递对象或者函数返回对象值

时，都会自动调用复制构造函数。

程序设计方法提示：有时候实现复制构造函数时不能只是简单地赋值，如对象中包含指向动态分配空间的指针数据成员时，简单地赋值会导致两个对象的指针数据成员指向同一块空间，这样很容易导致内存管理上的错误。对于这种情况的正确做法是，为当前新建对象的指针数据成员分配新的空间，然后将参数对象中指针数据成员所指的动态分配空间中的数据复制过来。这样，才能既保证两个对象一致，又让二者保持相互独立性。

【例 11-16】 为 Circle 类添加复制构造函数。

```
//文件 circle.h: 定义类 Circle
#if !defined __CIRCLE__H__
#define __CIRCLE__H__

#include <iostream.h>

class Circle {
public:
    //带默认参数的构造函数
    Circle(double a=5.0, double b=5.0, double c=5.0)
    {
        x=a;
        y=b;
        if (c>0)
            r=c;
        else
            r=5.0;
    }

    //复制构造函数
    Circle(Circle &c)
    {
        x=c.x;
        y=c.y;
        r=c.r;
    }

    void draw()
    {
        cout<<"Draw a circle at ("<<x<<","
            <<y<<"), with radius "<<r<<"."<<endl;
    }

private:
    double x, y;                    //x、y坐标
    double r;                       //半径
};

#endif
```

11.5 析构函数

析构函数也是类的一个特殊的成员函数。析构函数的函数名是字符~加上类名,和构造函数一样,析构函数也不能指定返回类型。从形式上来说,析构函数比构造函数更简单,它不能被重载,也没有参数。

11.5.1 析构函数的作用

从字面上理解,析构函数是构造函数的“反函数”,字符~在C语言中是按位取反运算符。实际上,从某种意义上来说,析构也正是构造的逆操作。构造函数的作用是创建对象时对对象进行初始化,它保证程序在使用对象时,对象中的数据已经处于正确的状态。析构函数的作用则是在撤销对象前,为对象做一些最后的清理工作,一般表现在释放对象在生存期内动态分配的空间。

如果类的对象不需要动态分配空间,如例11-16中定义的Circle类,则没有必要为类提供析构函数;而当类的对象包含动态分配的空间时,如例11-2中定义的Stack类(其栈空间是根据需要动态分配的)则必须使用析构函数。

对于析构函数,常见的程序设计错误是为析构函数提供参数、指定析构函数的返回类型或重载析构函数。

11.5.2 析构函数执行的时机

对象被撤销时,类的析构函数被自动调用。析构函数的执行不会破坏对象本身,它只是做些清理工作,如回收动态分配的空间。对象本身的回收工作由系统进行。对于在程序中不同地方声明的、具有不同作用域的对象,其析构函数执行的时机也各不相同。

全局对象存在于程序的整个生命周期,在所有程序执行完毕之后被撤销。在程序main函数的最后一条语句执行完毕后或程序执行exit语句之后,全局对象的析构函数才被执行。但如果程序以abort语句终止执行,全局对象的析构函数将不被执行。

局部动态对象在退出该对象的作用域(退出声明该对象的程序块)时被撤销,其析构函数也在退出该程序块时被执行。如果程序在退出该程序块之前以exit语句或abort语句终止执行,则局部动态对象的析构函数将不会被执行。

局部静态对象被创建后即存在于程序的整个生命周期,在程序执行完毕后或程序执行exit语句之后,局部静态对象的析构函数被执行,如果程序以abort语句终止执行,局部静态对象的析构函数将不被执行。局部静态对象的析构函数在全局对象的析构函数执行之前执行。

动态创建的对象(用new操作符创建)必须动态撤销,其析构函数将在执行相应的delete操作时执行。如果程序没有显式地使用delete操作符释放动态创建的对象,该对象将不会被释放,程序就会产生内存泄漏。

下面通过一个例子,综合理解全局对象、局部动态对象、局部静态对象和动态创建对象的构造函数与析构函数的执行时机。

【例11-17】 编写一个程序演示全局对象、局部动态对象、局部静态对象和动态创建对

象的构造函数与析构函数的执行时机。

```
//文件 createanddestroy.h
//定义类 CreateAndDestroy
#if !defined __CREATEANDDESTROY__H__
#define __CREATEANDDESTROY__H__
#include<iostream.h>
class CreateAndDestroy {
public:
    CreateAndDestroy (int n)
    {
        no=n;
        cout<<"Object"  <<no<<" created!"<<endl;
    }
    ~CreateAndDestroy()
    {
        cout<<"Object"<<no<<" destructed!"<<endl;
    }
private:
    int no;
};
#endif
//文件 ex11_17.cpp: 演示构造函数和析构函数的执行时机
#include "createAndDestroy.h"
CreateAndDestroy one(1);                     //定义全局对象 one
int main()
{
    CreateAndDestroy *ptr;
    CreateAndDestroy two(2);                 //定义局部动态对象 two
    {
        CreateAndDestroy three(3);           //定义局部动态对象 three
        ptr=new CreateAndDestroy(4);         //动态创建对象
        static CreateAndDestroy five(5);     //定义局部静态对象 five
    }
    delete ptr;                              //释放动态创建的对象
    return 0;
}
```

运行结果：

```
Object 1 created!
Object 2 created!
Object 3 created!
Object 4 created!
Object 5 created!
Object 3 destructed!
```

```
Object 4 destructed!
Object 2 destructed!
Object 5 destructed!
Object 1 destructed!
```

例11-17中的程序分为两个文件，头文件createAndDestroy.h定义了一个简单的类CreateAndDestroy，类CreateAndDestroy中包含一个私有数据成员no，存储对象的信息，构造函数和析构函数被执行时输出该信息以区分不同的对象。源文件ex11_6.cpp中分别在程序的不同地方声明和创建了类CreateAndDestroy的全局对象one、局部动态对象two和three、局部静态对象five和动态创建对象(no成员值为4)，程序的运行结果显示了不同的对象的构造函数和析构函数的执行时机。

习　题　11

11.1　填空：

(1) 在类中，定义构造函数的目的是________。

(2) 静态成员________对象的成员。

(3) 构造函数是和________同名的函数。

(4) 类是用户定义的类型，具有类类型的变量称为________。

(5) 当创建一个新对象时，程序自动调用________。

(6) 类包括两种成员：________和________。

11.2　编写Time类，要求：

(1) 包含年、月、日、时、分、秒的信息。

(2) 构造函数将类的对象初始化为系统当前时间(使用头文件time.h中的time()函数)。

(3) 能按标准格式输出对象表示的时间。

11.3　填空：

(1) 类test的析构函数名是________。

(2) 类的public成员函数集称为类的________。

(3) 类的构造函数________重载，类的析构函数________重载。

(4) 复制构造函数使用________来初始化创建中的对象。

(5) 指定为私有的类成员只能由类的________和________访问。

11.4　简述对象的初始化和对象的赋值有什么不同。

11.5　类和结构有何区别?

11.6　公有函数如何保护私有数据?

11.7　编写矩形类，要求可以提示用户输入矩形的长度和宽度，并显示该矩形的长度、宽度和面积。

11.8　完成下面类中的成员函数的实现。

```
class test{
private:
    int num;
    float f1;
public:
    test (int, float f);
    test (test&);
```

```
};
test:: test(int n, float f){
      num=n;
__________;
}
test:: test(__________) {
          __________;
          fl=t.fl;
};
```

11.9 定义分数类 Rational,要求在 private 部分用整数表示分子和分母,分子和分母以简化形式表示,即 24/36 应该以 2/3 的形式表示,并提供 public 成员函数实现如下功能:

(1) 两个分数相加,结果表示为简化形式。

(2) 两个分数相减,结果表示为简化形式。

(3) 按 a/b 的形式输出分数的值,a、b 为整数。

(4) 按浮点数的形式输出分数的值。

11.10 编写 HugeInt 类,要求用整型数组存放最多 30 位的整数值,并提供公有成员函数实现如下功能:

(1) 两个 HugeInt 对象相加,结果为 HugeInt。

(2) 两个 HugeInt 对象相减,结果为 HugeInt。

(3) 两个 HugeInt 对象相乘,结果为 HugeInt,要判断是否溢出。

(4) 两个 HugeInt 对象相除。

(5) 比较两个 HugeInt 对象的大小。

(6) 判断 HugeInt 对象是否大于 0、小于 0 或等于 0。

第 12 章 类和对象的使用

【学习内容】

本章介绍类和对象的使用中几个问题。主要内容包括：

- 类的复合。
- this 指针。
- const 对象和 const 成员函数。
- 友元函数。
- 友元类。

【学习目标】

- 掌握利用复合机制编写较复杂类的方法。
- 理解 this 指针的作用、原理，了解类和对象的实现机制。
- 掌握使用 this 指针编写能连续调用的成员函数的方法。
- 掌握 const 对象和 const 成员函数的定义和使用方法。
- 掌握编写友元函数的方法。
- 掌握友元类的用法。

上一章讨论了类和对象的基本概念，介绍了如何正确地定义一个类和建立类的实例——对象的方法。本章继续讨论类和对象的一些深层次的问题，包括类的复合、this 指针和友元，这些内容可以帮助程序员在程序设计中更好地使用类和对象。

12.1 类的复合

类的复合是软件重用的一种形式，是指在定义一个类时，把其他类的对象作为自己的成员。第 10 章介绍面向对象程序设计的概念时，曾经提到过对象和对象之间应该允许存在包含的关系，在现实生活中，这种需求无处不在。

下面定义一个描述圆柱的类，圆柱可以用底（一个圆）和高（一个浮点类型数据）两个属性来描述。既然前面已经有了圆类 Circle，那么为什么不直接把 Circle 的对象加入到圆柱中来呢？例 12-1 定义了一个圆柱体类 Column，并编写程序对类 Column 进行测试，类 Column 的数据成员之一 circle 是类 Circle 的实例化对象。程序在创建一个 Column 对象时，会同时创建一个 Circle 对象，在这个过程中会调用类 Column 和类 Circle 的构造函数。

需要注意程序中类 Column 的构造函数如何调用 Circle 构造函数，以及 Column 的对象如何访问 Circle 对象。

【例 12-1】 定义圆类 Circle，利用类的复合定义圆柱类 Column，然后创建圆柱类 Column 的实例化对象。

```
//文件 circle.h: 定义类 Circle
#if !defined __CIRCLE__H__
#define __CIRCLE__H__

#include <iostream.h>

class Circle {
public:
    Circle(double a=5.0, double b=5.0, double c=5.0);
    ~Circle();
    double area();
    void print();

private:
    double x, y;                              //x、y坐标
    double r;                                 //半径
};

#endif

//文件 circle.cpp: 类 Circle的实现部分
#include "circle.h"

#define PI 3.14159

Circle::Circle(double a, double b, double c)
{
    x=a;
    y=b;
    if (c>0)
        r=c;
    else
        r=5.0;

    cout<<"Circle object start: ";
    print();
    cout<<endl;
}

Circle::~Circle()
{
    cout<<"Circle object end: ";
    print();
    cout<<endl;
}

double Circle::area()
```

```
{
    return PI * r * r;
}
void Circle::print()
{
    cout<<"x="<<x<<", y="<<y<<", r="<<r;
}
//文件 column.h: 定义类 Column
#if!defined __COLUMN__H__
#define __COLUMN__H__
#include "circle.h"
class Column {
public:
    Column(double h=5.0, double a=5.0, double b=5.0, double c=5.0);
    ~Column();
    double volume();
    void print();
private:
    Circle circle;                          //数据成员 circle
    double height;                          //数据成员 height
};
#endif
//文件 column.cpp: 类 Column 的实现部分
#include "column.h"
Column::Column(double h, double a, double b, double c)
    :circle(a,b,c)                          //为数据成员 circle 调用其构造函数
{
    //初始化其他数据成员
    if (h>0)
        height=h;
    else
        height=5.0;
    cout<<"Column object start: ";
    print();
    cout<<endl;
}
Column::~Column()
{
    cout<<"Column object end: ";
    print();
    cout<<endl;
}
```

```
double Column::volume()
{
    return height * circle.area();
}
void Column::print()
{
    cout<<"height="<<height<<", ";
    circle.print();
}
//文件 ex12_1.cpp: 使用类 Column
#include "column.h"
#include <iostream.h>
int main()
{
    Column obj(2.3, 3.4, 4.5, 5.6);
    cout<<"The volume of obj is "<<obj.volume()<<endl;
    return 0;
}
```

运行结果：

```
Circle object start: x=3.4, y=4.5, r=5.6
Column object start: height=2.3, x=3.4, y=4.5, r=5.6
The volume of obj is 226.597
Column object end: height=2.3, x=3.4, y=4.5, r=5.6
Circle object end: x=3.4, y=4.5, r=5.6
```

例 12-1 中的程序包含了 5 个文件：类 Circle 的定义、类 Circle 的实现、类 Column 的定义、类 Column 的实现和主程序。Column 类包含两个私有属性：一个是类 Circle 的对象 circle(类的复合)；另一个是描述圆柱高度的属性 height。

声明类的对象时，其构造函数会被自动调用。如果一个类包含成员对象(circle 是类 Column 的成员对象)，成员对象会在包含它的对象创建之前被创建。如语句

```
Column column;
```

声明了类 Column 的一个对象 column，由于类 Column 中包含成员对象 circle，在对象 column 被创建之前，要先创建其成员对象 circle，类 Circle 的构造函数也会在类 Column 的构造函数执行之前，被类 Column 的构造函数调用。

如果成员对象的构造函数需要参数，这些参数也要通过复合类的构造函数传递进去。例 12-1 中的类 Circle 的构造函数就需要 3 个参数，因为在创建类 Column 的对象时，类 Circle 的构造函数是被复合类 Column 的构造函数调用的，这 3 个参数通过类 Column 的构造函数传入：类 Column 的构造函数接收到所有参数后，首先调用成员对象的构造函数，并传递相应的参数，然后再初始化自己的数据成员。程序语句如下：

```
Column::Column(double h, double a, double b, double c)
     :circle(a,b,c)                       //为数据成员 circle调用其构造函数
{
     //初始化其他数据成员
     …
}
```

类 Column 的构造函数有4个参数,在构造函数的头部多了冒号后面的部分,这部分实现了对数据成员 circle 的创建,调用成员对象的构造函数并传递参数。冒号后面是成员对象名(circle),对象名后面的参数列表提供了类 Circle 的构造函数所需的参数。如果 Column 类有多个成员对象,则可以在冒号后面分别列出各成员对象及其参数,中间用逗号隔开即可。这种直接指明成员对象名、并传递参数的方式称为成员对象构造函数的显式调用。如果成员对象的构造函数不需要参数(该成员对象所属的类有默认构造函数),则不需要在这里列出,但成员对象的默认构造函数一样会被调用,这种调用方法称为隐式调用。

在类 Column 的构造函数和析构函数中,都输出了类对象的圆心、半径和高度的值以标明该对象的身份。但是,圆心和半径都是成员对象 circle 的私有属性,在 Column 的构造函数和析构函数中无法直接访问,程序采用的方法是在 Circle 类中提供一个公有成员函数 print 用于输出这些属性,然后在类 Column 的构造函数和析构函数中调用该函数。

另外,应当注意到,类 Column 的析构函数并没有显式调用类 Circle 的析构函数。这是因为在对象复合方式下,在类 Column 的对象因撤销而执行析构函数时,会自动调用它的成员对象 circle 的析构函数。

类 Circle 的对象 circle 虽然是 Column 类的数据成员,但对于 circle 对象来说,Column 中的所有成员函数都是外部函数,仍然不能直接访问其私有属性。

类的复合将其他类的对象作为自己的成员对象,成员对象具有类作用域,可以被类的成员函数直接引用。

例12-1给出的是一个整体与部分的关系(圆是圆柱的一部分)的例子,很多时候为了管理的方便,也会使用类的复合。例如,前面例10-1给出了一个音像资料类 Media,为了更好地管理音像资料,可以定制一个个的资料架来存放这些音像资料,并记录每个资料架中存放了哪些音像资料,以及每个音像资料存放于哪个资料架的第几个位置。为此,需要定义一个资料架类 Shelf,假设每个资料架包含50个存放音像资料的位置,则可以在类 Shelf 中包含50个 Media 对象,例12-2给出了具体的实现方法。

【例12-2】 定义资料架类 Shelf,使每个资料架能管理50个音像资料,并编写测试程序。

```
//文件 media.h: 定义类 Media
#if !defined __MEDIA__H__
#define __MEDIA__H__
//定义媒体类型
#define ISAUDIO    1
#define ISVIDEO    2
class Media {
```

```
public:
    //构造函数
    Media();
    //析构函数
    ~Media();

    void setType(int t);
    void setName(char * n);
    void showInfo();

private:
    char * name;
    int type;
};

#endif

//文件 media.cpp: 类 Media 的实现部分
#include "media.h"
#include <iostream.h>
#include <string.h>

#define ISNULL 0

Media::Media()
{
    type=ISNULL;
    name=NULL;
}

Media::~Media()
{
    if (name !=NULL)
        delete [] name;
}

void Media::setName(char * n)
{
    if (name !=NULL)
        delete [] name;

    if (n !=NULL)
    {
        name=new char[strlen(n)+1];
        strcpy(name,n);
    }
    else
        name=NULL;
}

void Media::setType(int t)
{
```

```
        if (t==ISAUDIO || t==ISVIDEO)
            type=t;
}
void Media::showInfo()
{
    cout<<name<<" is a ";
    if (type==ISAUDIO)
        cout<<"audio disc.";
    else if (type==ISVIDEO)
        cout<<"video disc.";
    else
        cout<<"unclassified disc.";
    cout<<endl;
}
//文件 shelf.h: 定义复合类 Shelf
#if !defined __SHELF__H__
#define __SHELF__H__
#include "media.h"
#define MAX_MEDIA_NUM    50
class Shelf {
public:
    Shelf();
    int add(int t, char * n);                    //资料架增加一个媒体资料
    void list();                                 //列出资料架上所有的媒体资料
private:
    Media media[MAX_MEDIA_NUM];
    int num;
};
#endif
//文件 shelf.cpp: 类 Shelf 的实现部分
#include "shelf.h"
#include <iostream.h>
Shelf::Shelf()
{
    num=0;
}
//资料架增加一个媒体资料
int Shelf::add(int t, char * n)
{
    if (num>=MAX_MEDIA_NUM)
        return 0;
    media[num].setType(t);
```

```
        media[num].setName(n);
        num++;
        return 1;
    }
    //列出资料架上所有的媒体资料
    void Shelf::list()
    {
        int i;
        for (i=0; i<num; i++)
        {
            cout<<"No."<<i+1<<": ";
            media[i].showInfo();
        }
    }
    //文件 ex12_2.cpp: 使用类 Shelf
    #include "shelf.h"
    int main()
    {
        Shelf shelf;
        shelf.add(ISAUDIO, "The Colour of My Love");
        shelf.add(ISAUDIO, "Thriller");
        shelf.add(ISVIDEO, "Forrest Gump");
        shelf.add(ISVIDEO, "Slumdog Millionaire");
        shelf.list();
    return 0;
    }
```

运行结果：

```
No.1: The Colour of My Love is a audio disc.
No.2: Thriller is a audio disc.
No.3: Forrest Gump is a video disc.
No.4: Slumdog Millionaire is a video disc.
```

例 12-2 中的类 Shelf 中包含了类 Media 的一个数组 media，语句

```
Shelf shelf;
```

声明了类 Shelf 的对象 shelf，同时创建了类 Media 的 50 个对象，也调用了类 Media 的构造函数将这 50 个对象的数据成员 type 和 name 分别初始化为 ISNULL 和 NULL，表示该位置没有存放任何音像资料。语句

```
shelf.add(ISAUDIO, "The Colour of My Love");
```

调用类 Shelf 的成员函数 add 将音乐光盘“The Colour of My Love”放入资料架 shelf 中，并修改了资料架中相应位置的信息。语句

```
shelf.list();
```

则调用类 Shelf 的成员函数 list 输出资料架中存放的所有音像资料的信息。

例 12-2 只实现了对音像资料的存储管理和信息输出,在现有程序框架下要实现资料的删除、查询以及多资料架的管理都不复杂,读者可以自己动手试试。

12.2 this 指针

定义一个类就是定义一种抽象数据类型,一个类可以用来声明多个对象,所有对象都拥有类的非静态数据成员的一个副本,类的静态数据成员和成员函数则被所有对象共享,同一个类的不同对象响应相同的消息时,调用的是同一个函数。这里就有一个问题,当一个对象接收外界消息(调用对象的成员函数)时,被调用的成员函数如何保证它当前要操作的数据是这个对象的数据成员,而不会访问到其他对象呢?

答案是每个对象都维护了一个指向自身的指针,这个指针就是 this 指针。类的成员函数就是通过对象的 this 指针访问对象的数据成员的,this 指针是对象调用成员函数时的一个隐含参数。对象在调用其成员函数时,会将自己的 this 指针传递给成员函数,因为 this 指针不在成员函数的参数列表中列出,所以称 this 指针为隐含参数。当然,成员函数在获得对象的 this 指针后也可以显式的使用它,如可以通过 this 指针访问当前对象的成员或获取当前对象的地址等。

this 指针是属于对象的,每个对象都有属于自身的 this 指针,但类的定义中没有 this 指针。this 指针的类型由对象的类型决定,如 Circle 类的对象的 this 指针是 Circle * 类型。this 指针是常量指针,永远指向对象的首地址。所以,Circle 类的对象的 this 指针的类型是:Circle * const。

下面通过一个例子来说明 this 指针在成员函数中的作用,并在成员函数中显式地使用 this 指针来访问对象的数据成员。

【例 12-3】 编写程序用 this 指针访问对象的数据成员。

```
//文件 ex12_3.cpp:使用 this 指针访问数据成员
#include <iostream.h>

class Test {
public:
    Test(int n=0);
    void print();

private:
    int data;
};

Test::Test(int n)
{
    data=n;
}

void Test::print()
```

```
{
    cout<< "data= "<< this- > data<< "!"<< endl;
}
int main()
{
    Test obj1(10), obj2(20);
    cout<< "obj1:";
    obj1.print();
    cout<< "obj2:";
    obj2.print();
    return 0;
}
```

运行结果：

```
obj1:data= 10!
obj2:data= 20!
```

在例 12-3 中，类 Test 的成员函数 print 访问数据成员时不是像前面的例子那样直接使用数据成员的名字去访问，而是通过 this 指针加箭头运算符的方式，在执行这个函数时，this 的值是什么呢？前面说过，所有对象都维护着一个指向自身的指针，所以对象 obj1 的 this 指针的值就是它自身的首地址。当程序通过对象 obj1 调用成员函数 print(即执行语句 obj1. print())时，函数 print 会获得当前对象的 this 指针(即对象 obj1 的 this 指针)，然后 print 函数通过 this 指针去访问数据成员(即使代码中写的是通过数据成员名去直接访问，实际上也是通过 this 指针访问)。当一个成员函数响应消息时，它能够获得接收消息的对象的 this 指针，而成员函数访问数据成员时都是通过 this 指针去访问，所以总能访问到正确对象的数据成员。

this 指针还有一个作用，就是帮助对象获取对象自身，从而实现对成员函数的连续调用。因为 this 指针指向对象自身的首地址，指针复引用 * this 就能获得对象自身，如果一个类的成员函数返回值是 * this，程序就可以用这个返回值去调用另外一个成员函数。如语句

```
cout<< "data= "<< this- > data<< "!" << endl;
```

就是连续 4 次调用对象 cout 中的成员函数 operator<<(将在第 13 章详细讲述)。

【例 12-4】 定义一个类，要求其对象可以连续调用其成员函数，创建该类的对象并连续调用其成员函数。

分析：要满足类的对象可以连续调用其成员函数，这些成员函数的返回值就必须是调用该函数的对象本身，所以返回类型是该类型的引用，返回值则是 * this。

```
//文件 ex12_4.cpp: 使用 this 指针实现成员函数的连续调用
# include < iostream.h>
class Test {
public:
```

```
        Test(int n=0);
        Test & setData(int n);
        void print();
    private:
        int data;
    };
    Test::Test(int n)
    {
        data=n;
    }
    Test & Test::setData(int n)
    {
        data=n;
        return * this;
    }
    void Test::print()
    {
        cout<<"data="<<data<<"!"<<endl;
    }
    int main()
    {
        Test obj1;
        cout <<"obj1:";
        obj1.setData(100).print();
        return 0;
    }
```

运行结果：

```
obj1:data=100!
```

例12-4中，类Test的setData成员函数的返回类型是Test &，返回值是 * this，main函数中调用成员函数的语句是：

```
obj1.setData(100).print();
```

对象obj1连续调用了成员函数setData和print。通过obj1调用成员函数setData时，成员函数setData获得了obj1的this指针，函数setData执行完毕后，返回 * this，实际上就是obj1对象本身。所以，函数调用obj1.setData(100)的返回值是对象obj1的引用，因而可以通过点运算符继续调用该对象的成员函数，甚至可以接着调用setData函数。

如果函数调用的语句写成：

```
obj1.print().setData(100);
```

就会出错，因为对象obj1调用成员函数print之后获取的返回值不是obj1自身(或引用)，

不能用来接着调用类的成员函数。如果想实现连续调用多个函数，只要将需要连续调用的这些成员函数都定义成 setData 的形式（返回类型是 Test &，返回值是 * this）即可。

再回过头来分析静态成员函数和静态数据成员的问题。为什么类的静态成员函数不能访问非静态数据成员？因为静态成员函数是属于类的，即使在没有对象存在的情况下也可以通过类名去调用它们，静态成员函数不维护 this 指针，通过对象调用静态成员函数时也不将对象自己的 this 指针作为隐含参数传递过去。而非静态数据成员是属于每一个对象的，要访问某一个对象的数据成员必须通过该对象的 this 指针（或对象名）才能进行，而调用静态函数时，没有传入任何对象的 this 指针，所以静态成员函数就不能访问对象的非静态数据成员。静态成员函数也不能调用非静态成员函数，因为非静态成员函数必须维护 this 指针，而静态成员函数则没有，不能给非静态成员函数提供这个隐含参数。

12.3　const 特性

前面已经强调过，最低访问权限原则是保持良好程序设计风格的一项基本原则。本节要讲的就是该原则在对象应用上的体现。

有些对象是允许修改的，有些对象则不允许修改，有些成员函数需要修改对象的数据成员，另外一些成员函数则不需要修改对象的数据成员。在程序中可以用关键字 const 指定不允许被修改的对象或者不需要修改对象数据成员的成员函数。这样，任何试图修改 const 对象的操作在编译的时候都会报错，而 const 成员函数如果试图去修改数据成员也会被编译器发现并报错。

声明 const 对象时，关键字 const 在类名前面（和声明 const 变量相同），下面的语句声明了 Circle 类的一个 const 对象 baseCircle，并将它初始化为圆心坐标为（0，0），半径为 1 的一个圆。

```
const Circle baseCircle(0, 0, 1);
```

类的 const 成员函数不允许修改定义于类中的数据成员。声明 const 成员函数时，const 必须放在函数体之前，参数列表之后。例如，下面的语句声明了 const 成员函数 print。

```
void Test:: print() const
{
    cout<<"data="<<data<<"!\n";
}
```

C++ 对类和对象中的 const 声明是非常严格的。首先，如果一个对象被声明为 const 的，则不能通过该对象调用非 const 成员函数，因为非 const 成员函数有修改该对象的数据成员的可能。即使被调用的非 const 成员函数并不会修改对象，C++ 编译器也不允许这种调用。其次，类的 const 成员函数不允许显式或隐式地通过 this 指针调用非 const 成员函数，因为 const 成员函数不允许修改对象，而非 const 成员函数则有可能修改对象，const 成员函数如果调用了非 const 成员函数就有了间接修改对象的可能。总之，const 成员函数不能修改对象，const 对象不允许被任何程序修改。而通过非 const 对象，既可以调用非 const 成员函数，也可以调用 const 成员函数；非 const 成员函数也可以调用非 const 成员函数和 const 成员函数。

const 对象仅允许被构造函数和析构函数修改,与一般对象一样,构造函数可对 const 对象进行初始化,析构函数也同样可在撤销 const 对象前做些清除工作。

设置对象的 const 特性时常见的程序设计错误有:

- 试图修改 const 对象;
- 试图在 const 成员函数中修改对象的数据成员;
- 试图在 const 成员函数中调用非 const 成员函数;
- 试图通过 const 对象调用非 const 成员函数。

例 12-5 中的程序定义了 Circle 类的一个 const 对象,并试图通过该对象调用非 const 成员函数。如果 const 对象需要调用某些成员函数,必须把这些函数声明为 const 成员函数。

【例 12-5】 编写程序试图用 const 对象和 const 成员函数调用非 const 成员函数,并记录编译信息。

分析:本题可以借用前面例子中的 Circle 类,在其中定义一个 const 成员函数 print,并在 print 中调用非 const 成员函数 area,在主程序 main 函数中声明 Circle 类的 const 对象 obj,然后通过 obj 调用非 const 成员函数 area。

```
//文件 circle.h: 定义类 Circle
#if !defined __CIRCLE__H__
#define __CIRCLE__H__

#include <iostream.h>

class Circle {
public:
    Circle(double a=5.0, double b=5.0, double c=5.0);
    ~Circle();
    double area();
    void print() const;
    double getR() const;

private:
    double x, y;                    //x、y坐标
    double r;                       //半径
};

#endif

//文件 circle.cpp: 类 Circle的实现部分
#include "circle.h"

#define PI 3.14159

Circle::Circle(double a, double b, double c)
{
    x=a;
    y=b;
    if (c>0)
        r=c;
```

```
    else
        r=5.0;
    cout<<"Circle start, x="<<x<<", y="<<y<<", r="<<r<<endl;
}
Circle::~Circle()
{
    cout<<"Circle end, x="<<x<<", y="<<y<<", r="<<r<<endl;
}
double Circle::area()
{
    return PI * r * r;
}
void Circle::print() const
{
    double a=area();                    //错误语句
    cout<<"area="<<a <<endl;
}
double Circle::getR() const
{
    return r;
}
//文件 ex12_5.cpp: 使用类的 const 对象
#include "circle.h"
#include <iostream.h>
int main()
{
    const Circle obj;
    cout<<obj.area();                   //错误语句
    return 0;
}
```

编译结果：

```
ex12_5.cpp(8) : error C2662: 'area' : cannot convert 'this' pointer from 'const class Circle' to 'class Circle &'
circle.cpp(29) : error C2662: 'area' : cannot convert 'this' pointer from 'const class Circle' to 'class Circle &'
```

从例 12-5 的编译结果可以看出，在 Circle 类的 const 函数 print 中调用非 const 成员函数 area（文件 circle.cpp 第 29 行）是不允许的，错误原因是不能把 this 指针从 const class Circle 类型的指针转换成 class Circle & 类型的指针。this 指针的类型取决于类的类型，由于 this 指针是由对象维护的，指向对象本身，其类型和对象以及被调用的成员函数的 const 属性有关，对于 Circle 类的 const 对象，调用 const 成员函数时它的 this 指针的类型是 const

class Circle * const,调用非 const 成员函数时 this 指针的类型是 class Circle * const,因此,在 const 成员函数中调用非 const 成员函数时需要将 this 指针从 const class Circle * const 类型转换成 class Circle * const 类型,这是不允许的。在文件 ex12_5.cpp 的第 8 行,通过 const 对象调用非 const 成员函数 area 时也存在将 this 指针从 const class Circle * const 类型转换成 class Circle * const 类型的问题。

const 对象的初始化在构造函数中进行,初始化后的 const 对象不允许再被修改。const 关键字还可以作用于类的数据成员,const 数据成员必须在构造函数中初始化,但不能通过赋值的方式进行,初始化后的 const 数据成员也不能再被修改。例 12-6 演示了初始化 const 数据成员的方法。

【例 12-6】 初始化类的 const 数据成员。

```
//文件 circle.cpp: 初始化类的 const 数据成员
class Circle {
public:
    Circle(double a=5.0,double b=5.0,double c=5.0);
private:
    double x, y;                              //x、y坐标
    const double r;                           //半径
};
Circle:: Circle(double a, double b, double c)
      :r(c)                                   //const 数据成员的初始化
{
    x=a;
    y=b;
}
```

const 数据成员的初始化必须在构造函数的头部给出,由冒号引出 const 数据成员名和相应的初始化值,如果有多个 const 数据成员,则在冒号后面顺序列出,中间用逗号隔开。

如果试图用赋值语句初始化 const 数据成员,编译器就会报错。

例 12-7 演示了使用赋值语句初始化 const 数据成员的结果。

【例 12-7】 使用赋值语句初始化 const 数据成员。

```
//文件 circle.cpp: 初始化类的 const 数据成员
class Circle {
public:
    Circle(double a=5.0, double b=5.0, double c=5.0);
private:
    double x, y;                              //x、y坐标
    const double r;                           //半径
};
Circle:: Circle(double a, double b, double c)
{
    x=a;
    y=b;
```

```
    r=c;                                              //错误语句
}
```

编译结果：

```
error C2758: 'r' : must be initialized in constructor base/member initializer list
error C2166: l_value specifies const object
```

上面的错误信息显示，const 数据成员 r 必须在构造函数头部的初始化列表中进行初始化。

12.4 友元函数和友元类

类将数据成员和成员函数封装在一起，并通过访问说明符限定了成员的访问属性。类的对象的私有数据成员只能被类的成员函数访问。这种特性减少了程序出错的可能性，提高了程序的稳定性和质量。将类的数据成员放到 private 部分的前提是只有类的成员函数需要访问它。如果还有个别外部函数需要直接访问该数据成员呢？可以考虑把该数据成员放到 public 部分，虽然可以解决上述问题，但从保护数据的角度看来，这不是一个好方法。

另外一种方案是仍然把该数据成员放到 private 部分，然后再提供一种机制将那些允许访问这些数据成员的函数和其他不被允许的函数区分开，这种机制称为友元函数。也就是说，凡是声明为友元函数的外部函数可以访问这些私有成员，而其他外部函数则不能。本节将介绍这种友元机制，包括友元函数和友元类。

12.4.1 友元函数

一个类的友元函数是该类的外部函数，但它有权访问类的所有成员（包括私有成员和受保护成员）。

声明友元函数的方法是：在类的定义中加入该函数的函数原型，并将关键字 friend 放到函数原型前面。

友元函数的声明与类的访问说明符 private、protected 和 public 无关，可以放在类定义中的任何地方。

友元关系给程序设计在一定程度上提供了方便，但它也破坏了类的信息隐藏特性。

程序设计方法提示：建议一般不要使用友元，除非能带来极大的便利。

【例 12-8】 定义三角形类 Triangle，并使用友元函数修改其私有数据成员。

```
//文件 triangle.h: 定义类 Triangle
#if !defined __TRIANGLE__H__
#define __TRIANGLE__H__

class Triangle {
    friend void setA(Triangle &t, int n);          //声明 setA是本类的友元函数
public:
    Triangle(int x=5, int y=5, int z=5);
    void print();
```

```
private:
    int a, b, c;
};
#endif
//文件 triangle.cpp: 实现类 Triangle
#include "triangle.h"
#include <iostream.h>
Triangle::Triangle(int x, int y, int z)
{
    if (x+y>z && x+z>y && y+z>x)
    {
        a=x;
        b=y;
        c=z;
    }
    else
    {
        a=b=c=5;
    }
}
void Triangle::print()
{
    cout<<"Triangle: "<<a<<", "<<b<<", "<<c<<endl;
}
//文件 ex12_8.cpp: 使用友元函数
#include "triangle.h"
void setA(Triangle &t, int n)
{
    t.a=n;                              //访问对象 t 的私有成员 a
}
int main()
{
    Triangle t;
    t.print();
    setA(t, 10);
    t.print();
    return 0;
}
```

运行结果:

```
Triangle: 5, 5, 5
Triangle: 10, 5, 5
```

例 12-8 包含 3 个文件，分别为类 Triangle 的定义、类 Triangle 的实现和测试程序。类 Triangle 中第一条语句

```
friend void setA(Triangle &t, int n);
```

声明了外部函数 setA 为类 Triangle 的友元函数，为了能在函数 setA 中修改类 Triangle 的对象的数据成员，函数 setA 的第一个参数使用引用参数，并在其实现中通过对象名直接访问其私有数据成员。

12.4.2 友元类

一个类可以把另外一个类声明为自己的友元类，友元类的所有成员函数都可以直接访问该类的所有成员（包括私有成员和受保护成员）。

声明友元类的方法是：在类的定义中，在要声明的友元类的类名前面加上 friend 关键字。友元类的声明是单方面的，如类 A 声明 B 为自己的友元类，这并不意味着类 A 也是类 B 的友元类。友元类的声明也不是传递的，如类 A 声明类 B 是友元类，类 B 声明类 C 是友元类，这也不意味着类 C 就是类 A 的友元类。

【例 12-9】 定义两个类 A 和 B，将类 A 声明为类 B 的友元类，并在类 A 的成员函数中访问类 B 的私有数据成员，然后通过给类 A 的对象发消息来修改类 B 的对象的私有数据成员。

```
//文件 class.h: 定义类 A 和类 B
#if !defined __CLASS__H__
#define __CLASS__H__

class B;

class A {
public:
    void setB(B &b, int m);
    void print(B &b);
};

class B {
    friend class A;                             //声明友元类
private:
    int data;
};

#endif

//文件 class.cpp: 实现类 A 和类 B
#include "class.h"
#include <iostream.h>

void A::setB(B &b, int m)
{
    b.data=m;                                   //访问 B 的私有成员
```

```
}
void A::print(B &b)
{
    //访问 B的私有成员
    cout<<"The private data of class B: "<<b.data<<endl;
}
//文件 ex12_9.cpp: 使用友元类
#include "class.h"
int main()
{
    A a;
    B b;
    a.setB(b, 10);              //调用类 A的成员函数修改类 B的对象 b的私有数据
    a.print(b);                 //调用类 A的成员函数访问类 B的对象 b的私有数据
    return 0;
}
```

运行结果：

```
The private data of class B: 10
```

例12-9包含3个文件，其中文件class.h中定义了两个类：类A和类B。类B包含一个私有数据成员data，没有成员函数，类B中还包含一条友元声明语句：

```
friend class A;
```

该语句将类A声明为类B的友元类，这意味着类B允许类A的所有成员函数访问自己的所有成员(包括私有数据成员)。类A包含两个成员函数setB和print，setB修改类B的对象的私有数据成员，print函数则访问类B的对象的私有数据成员并将它输出到屏幕上。

最后再强调一次，除非带来极大的便利，不要使用友元。

习　题　12

12.1　填空：

(1) 友元函数________类的成员。

(2) 当一个成员函数被调用时，该成员函数的________指向调用它的对象。

(3) 友元函数可以存取对象的公有成员、________和________。

(4) 对象的成员函数可以通过________指针访问对象自身。

12.2　简述成员函数、全局函数和友元函数的差别。

12.3　对象为什么要维护一个this指针？

12.4　编写一个二叉树类，要求包含如下功能：

(1) 按中序遍历该二叉树。

(2) 在二叉树中查找指定的结点。

(3) 在二叉树中查找指定的结点所在的层次。

12.5 定义复数类 Complex,并使用友元函数实现复数的加法、减法、乘法,所有函数都返回 Complex 对象。

12.6 编写一个集合类,要求类的每个对象可以保存 0～100 个不同的整数,并使用友元函数实现如下功能:

(1) 往集合中加入一个整数。

(2) 从集合中去掉一个整数。

(3) 求两个集合的并集,结果是一个集合。

(4) 求两个集合的交集,结果是一个集合。

(5) 判断一个整数是否在集合中。

12.7 对如下圆柱类 Column 的定义:

```
class Column {
public:
    Column(double x, double y, double r, double h);
    ~Column();
    double area();                          //求圆柱面积
    double volume();                        //求圆柱体积
private:
    double height;                          //高度
    double radius;                          //半径
    double x;                               //底面圆心 x坐标
    double y;                               //底面圆心 y坐标
};
```

要求:

(1) 实现类 Column 中的 4 个成员函数。

(2) 增加一个友元函数,实现从键盘读取 4 个 double 类型的数据对类 Column 的对象进行定值的功能;

(3) 增加一个友元函数,实现将类 Column 的对象输出到屏幕的功能,输出信息包括对象的基本信息以及对象的面积和体积。

12.8 对于下面定义的类 Test:

```
class Test {
private:
        int a;
        double b;
};
```

要求:

(1) 为类 Test 的每个数据成员增加一个 set 函数,并使这些 set 函数都可以被连续调用。

(2) 为类 Test 的每个数据成员增加一个 get 函数。

(3) 如果要求所有 get 函数都可以被连续调用,该如何编写这些函数?

12.9 如果将你自己当作一个对象来看,那么复合、this 指针、const 属性、友元这些概念在自己身上怎么体现?

第 13 章 运算符重载

【学习内容】

本章介绍 C++ 的运算符重载机制。主要内容包括：

- ◆ 运算符重载的概念。
- ◆ 运算符重载的限制。
- ◆ 以成员函数或友元的形式重载单目运算符。
- ◆ 以成员函数或友元的形式重载双目运算符。
- ◆ 流插入和流提取运算符的重载。
- ◆ 赋值运算符的重载。
- ◆ 自定义类型与其他类型之间的转换。

【学习目标】

- ◆ 理解运算符重载的概念和意义。
- ◆ 理解运算符重载的限制。
- ◆ 掌握流插入和流提取运算符的重载方法。
- ◆ 掌握以成员函数的方式重载运算符的方法。
- ◆ 掌握以友元函数的方式重载运算符的方法。
- ◆ 理解运算符成员函数和友元函数的区别。
- ◆ 掌握赋值运算符重载的方法。
- ◆ 了解类型转换运算符的重载方法。
- ◆ 了解使用转换构造函数进行类型转换的方法。

类作为抽象数据类型实现了对数据和函数的封装，C++ 允许通过函数操作类的实例化对象。而对于 C++ 的内部类型，除了可以通过函数操作它们外，还可以用运算符实现它们之间的运算。运算符的使用，使程序的很多操作变得直观自然。

本章介绍如何将运算符作用于自定义类的对象上，包括如何以友元或成员函数的方式实现单目和多目运算符的重载，以及一些特殊运算符的重载，还包括流插入和流提取运算符、赋值运算符、类型转换运算符的重载等。

13.1 运算符重载的概念

在 C++ 中允许定义多个同名函数来实现类似的操作，只要这些函数的参数不同（参数的个数、类型或顺序不同），这种功能称为函数重载（function overloading）。运算符也可以

看成是一种函数，也能够被重载。其实，前面已经接触过很多运算符重载的实例。例如，+既可作为一元正运算符使用，也可作为二元加法运算符使用，而作加法运算符使用时既可实现两个 long 类型的数相加，又可实现两个 double 类型的数相加。也就是说，加法运算符既可以施加于整数上，也可以施加于实数上，虽然这两种“相加”功能的内部实现完全不同，但都使用了相同的运算符，与人们的数学表示习惯相符合。

将运算符作用于对象上的例子前面也接触过。例如，运算符<<既可以作为移位运算符，也可以作为流插入运算符，作为流插入运算符时，<<实际上是和对象一起用的。例如：

```
cout<< "String";
cout<< 8;
```

其中，cout 是输出流对象，上述语句就是在给 cout 发消息，响应消息的函数名是 operator<<。这里也对运算符<<进行了重载，既可以通过<<输出字符串，也可以输出整数，它们的内部实现也是不一样的。

13.1.1 运算符重载的意义

用运算符操作对象，实际上也是通过定义相应的函数来实现的，从根本上来说，用运算符操作对象和调用相应的函数没什么不一样。那么，运算符重载有什么意义呢？

从功能上讲，通过运算符能实现的功能通过函数一样能够实现，所以运算符重载没有在程序的功能上带来好处。

一般在编写程序时，对于系统内部类型（如整型、浮点类型等）的操作，能用运算符的时候人们都在尽量用，因为运算符的使用能使程序变得非常清晰、简单，符合人们的习惯。虽然 C++ 不允许定义新的运算符，但运算符重载允许将现有的运算符和自定义类型一起使用，从而实现更简洁、自然的表示方法，这无疑是 C++ 功能强大的特点之一。运算符重载使 C++ 具有更好的可扩充性，这也是 C++ 最具吸引力的特点之一。

运算符重载也不是在任何时候都是好的。在完成同样功能的情况下，如果使用运算符比使用一般的函数调用能使程序更清晰，那么就应该使用运算符重载。但是，过度地或不恰当地使用运算符重载，可能反而会使程序变得不清晰和难以理解。

运算符的重载必须通过编写函数来完成，运算符作用于对象上的功能由程序员自己确定。重载运算符的函数名是关键字 operator 加运算符，如重载运算符 + 的函数名为 operator+。

【例 13-1】 定义复数类 Complex，并使用运算符重载的方式实现复数的加、减和赋值运算。

```
//文件 complex.h: 复数类的定义
#ifndef __COMPLEX__H__
#define __COMPLEX__H__

class Complex
{
public:
    Complex(double=0.0, double=0.0);
    Complex operator+ (const Complex &) const;
```

```
    Complex operator- (const Complex &) const;
    Complex &operator= (const Complex &);
    void print() const;
private:
    double real;                        //实数部分
    double imaginary;                   //虚数部分
};
#endif
```

```
//文件 complex.cpp: 复数类的实现
#include <iostream.h>
#include "complex.h"
Complex::Complex(double r, double i)
{
    real= r;
    imaginary= i;
}
void Complex::print() const
{
    cout<< '('<< real<< ", "<< imaginary<< ')';
}
Complex Complex::operator+ (const Complex &operand2) const
{
    Complex sum;
    sum.real= real+ operand2.real;
    sum.imaginary= imaginary+ operand2.imaginary;
    return sum;
}
Complex Complex::operator- (const Complex &operand2) const
{
    Complex diff;
    diff.real= real- operand2.real;
    diff.imaginary= imaginary- operand2.imaginary;
    return diff;
}
Complex &Complex::operator= (const Complex &right)
{
    real= right.real;
    imaginary= right.imaginary;
    return * this;
}
```

```
//文件 ex13_1.cpp: 主函数定义,通过运算符操作复数对象
#include <iostream.h>
```

```
#include "complex.h"
int main()
{
    Complex x, y(4.3, 8.2), z(3.3, 1.1);
    //输出 x,y,z
    cout<<"x: ";
    x.print();
    cout<<"\ny: ";
    y.print();
    cout<<"\nz: ";
    z.print();
    //输出加法运算
    x=y+z;
    cout<<"\n\nx=y+z:\n";
    x.print();
    cout<<"=";
    y.print();
    cout<<"+";
    z.print();
    //输出减法运算
    x=y-z;
    cout<<"\n\nx=y-z:\n";
    x.print();
    cout<<"=";
    y.print();
    cout<<"-";
    z.print();
    cout<<'\n';
    return 0;
}
```

运行结果：

```
x: (0, 0)
y: (4.3, 8.2)
z: (3.3, 1.1)
x=y+z:
(7.6, 9.3)=(4.3, 8.2)+(3.3, 1.1)
x=y-z:
(1, 7.1)=(4.3, 8.2)-(3.3, 1.1)
```

例 13-1 的类 Complex 定义了 3 个特殊的成员函数（运算符重载函数）：operator＋、operator－和 operator＝。这 3 个函数分别实现了复数的加、减和赋值运算，而且因为它们都是运算符重载函数，可以使用运算符来操作复数对象。例如：

```
x=y+z;
x=y-z;
```

其中的运算符＋、－、＝实际上都是通过成员函数 operator＋、operator－ 和 operator＝实现，其功能也是由这3个函数的具体实现决定。也就是说，语句

```
x=y+z;
```

实际上调用了 operator+和 operator=两个函数，可以等价地改写为：

```
x.operator=(y.operator+(z));
```

从上面的例子可以看出，虽然运算符重载在功能上没有带来什么好处，但却可以增加程序的可读性，使程序变得更加直观。很显然，人们更愿意看到形如 x＝y＋z；的语句，而不是 x. operator＝(y. operator＋(z))；。

为了提高程序的可读性，使用运算符重载时，要注意保持重载运算符在语义上与运算符的原始经典语义的相似性，使重载的运算符作用于对象时的功能与作用于内部类型的功能尽可能相似。例 13-1 将运算符＋的功能实现为两个复数相加。如果在例 13-1 中将运算符＋的重载函数的功能实现为两个复数对象相减或实现复数对象的输出，编写出来的程序就会让人迷惑不解。因此，只有保持了重载运算符在语义上与运算符的原始经典语义的相似性，才能提高程序的可读性，这才是使用重载机制的初衷。

作用于自定义类的对象上的运算符大都必须重载，但有两种例外情形：一种情形是赋值运算符(＝)不需要重载就可以用在每一个类上，它实现的功能与例 13-1 中的重载赋值运算符函数类似，完成对应数据成员的赋值；另外一个不需要重载就可以直接使用的运算符是取地址符(&)，它可以返回对象在内存中的地址。

13.1.2　运算符重载的限制

C++中大部分运算符都可以被重载，但也有一些运算符是不允许重载的。表 13-1 和表 13-2 分别列出了可以被重载的和不能被重载的运算符。

表 13-1　C++中可以被重载的运算符

+	−	*	/	%	^	&	|
~	!	=	<	>	+=	−=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
−−	−>*	,	−>	[]	()	new	delete

表 13-2　C++中不能被重载的运算符

.	.*	::	?:	sizeof

出于程序可读性上的考虑，C++要求在重载运算符时不能改变运算符的属性，对运算符重载有如下限制。

(1) 重载运算符不能改变运算符的优先级。C++已经规定了所有运算符的优先级，一个运算符不管作用于什么对象上，它都具有固定的优先级，但可以使用圆括号改变运算符执

行的先后次序。

(2) 重载运算符不能改变运算符的结合性。有些运算符是左结合的,有些运算符是右结合的,运算符重载不能改变它们的结合性。

(3) 不能使用默认参数重载运算符。运算符重载函数不能使用默认参数,使用运算符时必须明确地指出其每一个操作数。

(4) 重载运算符不能改变运算符操作数的个数。C++ 中有单目运算符,也有双目运算符,单目运算符只能带一个操作数,双目运算符必须带两个操作数。运算符重载不能改变运算符所带操作数的个数。有些运算符(如 *)既是单目的(作为间接访问运算符时),也是双目的(作为乘号时),一次重载只能使用其中一种含义。

(5) 不能建立新的运算符。只能重载表 13-1 中列出的运算符,不能发明新的运算符进行重载,也不能重载表 13-2 中列出的运算符。

(6) 重载运算符时不能改变运算符作用于内部类型时的含义。运算符重载不能修改其作用于内部类型时的含义。例如,+作用于整型时表示两个整数相加,而且相加的结果也是一定的,程序员不能编写一个+重载函数,使得+作用于两个整数时产生另外的效果。

(7) 运算符重载必须作用于自定义类型。运算符重载函数可以作为类的成员函数重载,也可以作为类的友元函数重载,不管哪种情况都必须至少有一个操作数是自定义类型的对象。运算符重载必须是将运算符和自定义类型的对象联系在一起。

(8) 重载运算符()、[]、->、=时,重载函数必须声明为类的成员函数。

(9) 运算符都必须显式重载。

13.2 运算符成员函数与友元函数

运算符重载函数要求至少有一个参数是自定义类的对象。为了能操作对象的数据成员,运算符函数可以是类的成员函数,也可以是类的非成员函数。作为非成员函数时,一般是类的友元函数。重载为类的成员函数时,运算符函数通过类的对象调用,这种情况下,运算符函数可以通过 this 指针直接操作对象的数据成员,这个 this 指针就是类的对象传递给运算符函数的隐含参数。重载为友元函数时,运算符函数要操作的对象必须在参数列表中明确列出。

虽然运算符函数可以是成员函数,也可以是友元函数,但在表达式中使用该运算符操作对象时格式是一样的。

当运算符作为成员函数重载时(例 13-1 的 3 个运算符都重载为成员函数),它的左操作数(如果是单目运算符,则指其唯一的操作数)必须是成员函数所在类的一个对象,也就是调用该运算符函数的对象。如果左操作数是其他类的一个对象或内部类型的对象,则运算符必须作为非成员函数(一般是友元函数)重载。

定义运算符重载函数时,到底是使用成员函数还是友元函数呢?建议尽量使用成员函数的形式重载运算符,因为这样不会破坏类的封装和信息的隐藏,程序的结构会更清晰,而且实现起来也更简单。但有些情况下必须使用友元函数来重载运算符。

如果重载单目运算符,由于单目运算符只能带一个操作数,并且该操作数必须是自定义类的对象,这种情况下一般使用成员函数的形式重载。

对于有些双目运算符,如流插入运算符(<<),它的左操作数要求是对象cout,是其他类的一个对象,如果需要将<<和自定义类的对象一起使用(通过<<输出该类的对象),就必须以友元函数的形式重载运算符<<,因为当运算符作为成员函数重载时,它的左操作数必须是成员函数所在类的一个对象。

还有一种情况必须使用友元函数的形式重载运算符,就是需要保留某些运算符的可交换性时。例如,运算符+具有可交换性,程序希望重载+以实现自定义类的对象和整数相加,即实现语句obj1=obj2+1的操作。如果需要保留+的可交换性,既能接受表达式obj1=obj2+1,也能接受表达式obj1=1+obj2,就必须以友元函数的形式重载运算符+。如果以成员函数的形式重载,那么运算符+的左操作数只能是类的对象,就不能交换+的两个操作数了。

13.3 单目运算符重载

重载单目运算符时,其唯一的操作数只能是自定义类的对象或对象的引用。单目运算符可以作为类的成员函数重载,也可以作为类的友元函数重载。作为成员函数重载时没有参数,唯一的操作数就是调用该函数的对象;作为友元函数重载时必须带一个参数,参数为自定义类的对象或对象的引用。

下面定义一个字符串类String,并重载单目运算符!,如果s是一个字符串类的对象,则!s将用来测试s是否为空串(如果s是空串则返回真,否则返回假)。分别以成员函数和友元函数的形式重载。

【例13-2】 定义字符串类String,并以成员函数的方式重载运算符!以判断对象中的字符串是否为空串。

```
//文件 string.h: 定义类 String
#if !defined __STRING__H__
#define __STRING__H__
class String
{
public:
    String(char * m=NULL);                    //使用默认参数的构造函数
    ~String();
    //运算符重载成员函数原型
    bool operator !();
private:
    char * str;
};
#endif
//文件 String.cpp: 类 String 的实现
#include <string.h>
#include "string.h"
```

```
String::String(char * m)
{
    if (m==NULL)
    {
        str=NULL;
    }
    else
    {
        str=new char[strlen(m)+1];
        strcpy(str, m);
    }
}
String::~String()
{
    if (str !=NULL)
        delete []str;
}
//实现运算符重载函数
bool String::operator !()
{
    if (str==NULL || strlen(str)==0)
        return true;
    return false;
}
//文件 ex13_2.cpp: 以成员函数的方式重载运算符!
#include <iostream.h>
#include "String.h"
int main()
{
    String s1, s2("some string");
    if (!s1)
        cout <<"s1 is NULL!"<<endl;
    else
        cout <<"s1 is not NULL!"<<endl;
    if (!s2)
        cout <<"s2 is NULL!"<<endl;
    else
        cout <<"s2 is not NULL!"<<endl;
    return 0;
}
```

运行结果：

```
s1 is NULL!
```

```
s2 is not NULL!
```

重载运算符!的函数名为 operator!,由于!是单目运算符,作为成员函数重载时,重载函数没有参数(唯一的操作数就是调用该函数的对象)。定义完运算符函数后,可以使用两种方式调用该函数。一种是调用类的成员函数的方法,如判断类 String 的对象 s1 是否为空的语句为:

```
s1.operator!();
```

它通过对象 s1 调用其成员函数 operator!。另一种方法是表达式的方法,可写成

```
!s1;
```

两种方法在功能上完全等价,都相当于通过对象 s1 调用其成员函数 operator!。

运算符!还可以重载为友元函数,例 13-3 用友元函数的方式重载了运算符!,实现了和例 13-2 中的程序同样的功能。

【例 13-3】 定义字符串类 String,并以友元函数的方式重载运算符!判断对象中的字符串是否为空串。

```
//文件 string.h: 定义类 String
#if !defined __STRING__H__
#define __STRING__H__
class String
{
    //运算符重载友元函数
    friend bool operator !(const String &s);
public:
    String(char * m=NULL);
    ~String();
private:
    char * str;
};
#endif
//文件 String.cpp: 类 String 的实现
#include <string.h>
#include "string.h"
String::String(char * m)
{
    if (m==NULL)
        str=NULL;
    else
    {
        str=new char[strlen(m)+1];
        strcpy(str, m);
    }
```

```
}
String::~String()
{
    if (str != NULL)
        delete []str;
}
//实现运算符重载友元函数
bool operator !(const String &s)
{
    if (s.str==NULL || strlen(s.str)==0)
        return true;
    return false;
}
//文件 ex13_3.cpp: 以友元函数的方式重载运算符!
#include <iostream.h>
#include "String.h"
int main()
{
    String s1, s2("some string");
    if (!s1)
        cout <<"s1 is NULL!"<<endl;
    else
        cout <<"s1 is not NULL!"<<endl;
    if (!s2)
        cout <<"s2 is NULL!"<<endl;
    else
        cout <<"s2 is not NULL!"<<endl;
    return 0;
}
```

运行结果：

```
s1 is NULL!
s2 is not NULL!
```

作为友元函数重载时，运算符函数的函数名仍然是 operator!，由于是外部函数，其所有操作数都必须以参数的形式显式地列出来。所以，作为友元函数重载的运算符函数 operator!就必须带一个参数，是类 String 对象的引用。与以成员函数的方式重载运算符相似，也可以使用两种不同的方法来调用这个友元运算符函数。一种是通过函数调用方式：

```
operator!(s1);
```

该语句以对象 s1 为参数直接调用函数 operator!。另一种方式是采取表达式的方式，相应的语句为

```
!s1;
```

它相当于以对象 s1 为参数调用函数 operator!。

从上面的两个例子可以看到,作为成员函数重载和作为友元函数重载,这两种方法虽然在实现上不同,但是如果采用表达式的方式调用时,其形式与一般运算符表达式的语法完全一样,都是!s1,这会使编写出来的程序更加直观自然、简洁。

C++ 有两个特殊的单目运算符,自增和自减运算符(其实是 4 个,包括前置和后置的自增、自减运算符),它们都可以被重载。重载前自增运算符和后自增运算符时,其运算符函数名都是 operator++,为了既能重载前自增运算符,又能重载后自增运算符,C++ 在运算符函数的参数列表上对它们进行了区分。

重载前自增或前自减运算符时,其参数列表与一般单目运算符重载函数相同,即重载为成员函数时,运算符函数不带参数,重载为友元函数时,运算符函数带一个参数,其参数为该类的对象或对象的引用。

重载后自增或后自减运算符时,其参数列表与一般单目运算符重载函数不同。重载为成员函数时,运算符函数带一个整型参数(这是一个伪参数,其作用仅是将该函数与前自增或前自减运算符函数区分开);重载为友元函数时,运算符函数则带两个参数,第一个参数为该类的对象或对象的引用,第二个参数是作为伪参数的整型参数。

例如,如果要给类 Complex 的对象 c1 实现操作++c1,则运算符++需要作为前自增运算符重载,重载为成员函数时,其原型为:

```
Complex operator++();
```

编译器会将表达式++c1 转换成为:

```
c1.operator++();
```

如果重载为友元函数,则它在类 Complex 中的声明为:

```
friend Complex operator++(Complex &);
```

编译器会将表达式++c1 转换成为:

```
operator++(c1);
```

如果要给类 Complex 的对象 c1 实现操作 c1++,则运算符++就要作为后自增运算符重载。重载为成员函数时,其函数原型为:

```
Complex operator++(int);
```

编译器会将表达式 c1++转换成为:

```
c1.operator++(0);
```

这里的参数 0 是编译器自动加上的,它不参与任何计算,只是使该函数区别于前自增运算符的重载函数。

如果重载为友元函数,则其在类 Complex 中声明的函数原型为:

```
friend Complex operator++(Complex &, int);
```

相应地，编译器会将表达式 c1++转换成为：

```
operator++(c1, 0);
```

这里的 0 也是一个伪参数，不参与任何计算。

13.4 重载流插入和流提取运算符

在考虑双目运算符的重载之前，先看流插入运算符和流提取运算符。语句

```
cout<<"string";
```

能在标准输出设备上输出字符串"string"。在这条语句中，<<是一个双目运算符，cout 是类 ostream 的一个对象（定义于头文件 ostream.h 中）。在类 ostream 的定义中，可以看到很多重载运算符<<的成员函数 operator <<。程序每使用一次流插入运算符，实际上就是调用 ostream 类的一个成员函数。对象 cout 为什么可以输出所有的内部类型的对象？从类 ostream 的定义可以看出，实际上是因为在类 ostream 中都有相应的重载函数。如果程序员想用对象 cout 和流插入运算符（<<）输出自己定义的类型（如例 13-2 中定义的类 String）的对象，如何实现呢？怎么能使下面的语句按照要求的格式输出自定义类的对象呢？例如：

```
String s("string");
cout<<s <<endl;
```

要实现上面的语句，必须进行运算符重载，如何定义重载函数呢？考虑运算符<<的两个操作数，cout 和 s 分别是类 ostream 和类 String 的对象。要使语句 cout << s 合法有两种可能性：一种是将运算符函数 operator <<定义为类 ostream 的成员函数或友元函数；另一种方法是将该函数定义为类 String 的成员函数或友元函数。类 ostream 定义在系统头文件中，其具体实现无法修改，而定义为类 ostream 的成员函数或友元函数都必须修改 ostream 类，所以这种方法不可能也不可取。考虑类 String，由于运算符<<的最左操作数是 cout，不是类 String 的对象，因此该运算符不能作为类 String 的成员函数重载。这样，就只能考虑以类 String 的友元函数的形式重载，而且重载函数必须有两个参数：一个参数为类 ostream 的对象（cout）或对象引用，另一个参数为类 String 的对象或对象引用。由于流插入运算符是可以连续调用的，函数的返回类型必须为类 ostream 的对象（cout）的引用。

基于同样的考虑，重载流提取运算符（>>）的方法和重载流插入运算符的方法一样。由于其左操作数是类 istream 的对象 cin，所以也必须以类 String 的友元函数的形式重载。其重载函数也必须有两个参数：一个参数是类 istream 的对象或对象引用，另一个参数是类 String 的对象或对象引用。函数的返回类型必须为 istream 类对象（cin）的引用。

【例 13-4】 定义字符串类 String，并重载流插入运算符（<<）和流提取运算符（>>），实现用 cout 和 cin 直接输出和输入类 String 的对象。

```
//文件 string.h: 定义类 String
#if !defined __STRING__H__
#define __STRING__H__
```

```
#include <iostream.h>
class String
{
    //流插入运算符 <<重载函数
    friend ostream & operator<< (ostream &output, const String &s);
    //流提取运算符 >>重载函数
    friend istream & operator>> (istream &input, String &s);
public:
    String(char * m=NULL);
    ~String();
private:
    char * str;
};
#endif
//文件 String.cpp:类 String的实现
#include <string.h>
#include "string.h"
#define MAX_STR_LEN 256
String::String(char * m)
{
    if (m==NULL)
        str=NULL;
    else
    {
        str=new char[strlen(m)+1];
        strcpy(str, m);
    }
}
String::~String()
{
    if (str !=NULL)
        delete []str;
}
//实现运算符<<重载函数
ostream & operator<< (ostream &output, const String &s)
{
    if (s.str !=NULL)
        output<< s.str;
    return output;
}
//实现算符>>重载函数
istream & operator>> (istream &input, String &s)
{
```

```
        char temp[MAX_STR_LEN];
        input>> temp;
        if (s.str != NULL)
            delete []s.str;
        s.str= new char[strlen(temp)+ 1];
        strcpy(s.str, temp);
        return input;
}
//文件 ex13_4.cpp: 测试重载的运算符
#include <iostream.h>
#include "string.h"
int main()
{
        String s1, s2;
        cout<< "Please input two strings:"<< endl;
        cin>> s1>> s2;
        cout<< "Output is:"<< endl;
        cout<< "s1- - "<< s1<< endl<< "s2- - "<< s2<< endl;
        return 0;
}
```

运行结果：

```
Please input two strings:
Wuhan
CHangsha
Output is:
s1- - Wuhan
s2- - CHangsha
```

流插入运算符和流提取运算符的重载函数被定义为类 String 的友元函数。当程序执行到 main 函数中的语句 cin>>s1>>s2 时，首先是计算表达式 cin>>s1，产生如下的函数调用：

```
operator>> (cin, s1);                    //自定义重载函数
```

该函数从标准输入设备读取一个字符串，并将该字符串放到对象 s1 中。函数 operator>> 返回对象 cin 的引用，语句 cin>>s1>>s2 剩下部分就会产生如下函数调用：

```
operator>> (cin, s2);                    //自定义重载函数
```

这次调用从标准输入设备读取一个字符串放到对象 s2 中。语句

```
cout<< "s1- - "<< s1<< endl<< "s2- - "<< s2<< endl;
```

可以分解为如下函数调用序列：首先表达式 cout<<"s1——"调用定义于类 ostream 中的成员函数 operator<<，在标准输出设备上输出字符串"s1——"，然后返回对象 cout 的引用，

接下来表达式 cout<<s1 产生函数调用：

```
operator << (cout, s1);                    //自定义重载函数
```

该函数将对象 s1 中的数据 str 输出到标准输出设备上，并返回对象 cout 的引用，接着剩下部分会产生函数调用：

```
cout.operator<< (endl);                    //类 ostream 中定义的重载函数
cout.operator<< ("s2--");                  //类 ostream 中定义的重载函数
operator << (cout, s2);                    //自定义重载函数
cout.operator<< (endl);                    //类 ostream 中定义的重载函数
```

可以看出，程序员自定义的流插入和流提取运算符重载函数可以和系统重载的运算符一起使用，而且在格式上看不出任何差别，充分展现了 C++ 良好的可扩展性。

13.5 双目运算符重载

上一节讨论了两个特殊的双目运算符——流插入和流提取运算符的重载，本节讨论一般双目运算符的重载问题。重载双目运算符时要考虑两个因素：一是被重载的运算符对左操作数是否有要求；二是是否要保留运算符的可交换性(只针对具有可交换性的运算符而言)。双目运算符可以被重载为带有一个参数的非静态成员函数或带有两个参数的非成员函数，重载为非成员函数时其参数之一必须是类的对象或对象的引用。

例 13-1 以成员函数的方式重载了运算符+、-和=，实现了两个复数对象的加、减和赋值运算。例 13-5 针对类 String 实现双目运算符+=的重载，实现字符串的连接，即针对类 String 的对象 x、y，表达式 x+=y 的语义是：将对象 y 中的字符串连接到对象 x 的字符串后面，并将连接后的对象 x 作为运算的结果。

考虑表达式 x+=y，运算符+=的左操作数是类 String 的对象，而且+=不具有可交互性，所以运算符+=可以作为类 String 的成员函数重载，也可以作为友元函数重载。作为成员函数重载时，重载函数有一个参数，是类 String 的对象或引用，作为友元函数重载时，重载函数有两个参数，两个参数都是类 String 类型的对象或引用。

【例 13-5】 定义字符串类 String，并重载运算符+=，针对类 String 的对象 x、y，表达式 x+=y 的语义是：将对象 y 中的字符串连接到对象 x 的字符串后面，并将连接后的对象 x 作为运算的结果。

```
//文件 string.h: 定义类 String
#if !defined __STRING__H__
#define __STRING__H__

#include <iostream.h>

class String
{
    //运算符<<重载函数
    friend ostream & operator<< (ostream &output, const String &s);
    //运算符>>重载函数
```

```
    friend istream & operator>> (istream &input, String &s);
public:
    String(char * m=NULL);
    ~String();
    String & operator+= (const String &s);
private:
    char * str;
};
#endif
//文件 String.cpp: 类 String的实现
#include <string.h>
#include "string.h"
#define MAX_STR_LEN 256
String::String(char * m)
{
    if (m==NULL)
    {
        str=NULL;
    }
    else
    {
        str=new char[strlen(m)+1];
        strcpy(str, m);
    }
}
String::~String()
{
    if (str !=NULL)
        delete []str;
}
//实现运算符<<重载函数
ostream & operator<< (ostream &output, const String &s)
{
    if (s.str !=NULL)
        output<< s.str;
    return output;
}
//实现运算符>>重载函数
istream & operator>> (istream &input, String &s)
{
    char temp[MAX_STR_LEN];
    input>> temp;
```

```
        if (s.str != NULL)
            delete []s.str;
        s.str= new char[strlen(temp)+ 1];
        strcpy(s.str, temp);
        return input;
}
//实现运算符+=重载函数
String & String::operator+ = (const String &s)
{
        char * temp;
        temp= new char[strlen(str)+ strlen(s.str)+ 1];
        strcpy(temp, str);
        strcat(temp, s.str);
        if (str != NULL)
            delete []str;
        str= temp;
        return * this;
}
//文件 ex13_5.cpp: 测试重载的运算符
# include < iostream.h>
# include "string.h"
int main()
{
        String s1, s2;
        cout<< "Please input two strings:"<< endl;
        cin>> s1>> s2;
        cout<< "s1- - "<< s1<< endl<< "s2- - "<< s2<< endl;
        s1+ = s2;
        cout<< "after s1+ = s2, s1- - "<< s1<< endl;
        return 0;
}
```

运行结果:

```
Please input two strings:
Wuhan
Changsha
s1- - Wuhan
s2- - Changsha
after s1+ = s2, s1- WuhanChangsha
```

例13-5中的程序以成员函数的方式重载了运算符+=。表达式 x+=y 被处理成 x.operator+=(y),运算符函数 operator+=将对象 y 中的字符串拼接到对象 x 的字符串

后面。为了实现运算符+=的连续操作,运算符函数operator+=的返回类型被定义为String类型的引用,返回运算完成后的第一个操作数。运算符+=也可以重载为友元函数,重载为友元函数时,表达式 x+=y 会被处理成 operator+=(x,y),友元函数重载的定义及实现见例13-6。

以友元函数的方式重载运算符+=时,运算符函数 operator+=要带两个参数,两个操作数都必须在参数列表中明确列出。

【例 13-6】 以友元函数的形式重载运算符+=。

```
class String
{
    //声明友元函数 operator+=
    friend String &operator+= (String &x, String &y);
    ...
};
//实现运算符+=重载函数
String & operator+= (String &x, const String &y)
{
    char * temp;
    temp= new char[strlen(x.str)+ strlen(y.str)+ 1];
    strcpy(temp, x.str);
    strcat(temp, y.str);
    if (x.str != NULL)
        delete []x.str;
    x.str= temp;
    return  x;
}
```

13.6 赋值运算符重载

在例13-1中已经实现了赋值运算符(=)的重载,其实赋值运算符不用重载就可以直接用在自定义类的对象之间,其默认操作是逐个复制对象的所有数据成员。这种方式对大多数类的对象是合适的(包括例13-1中的类Complex),但如果对象中包含动态分配的空间时,这种赋值方式就有可能出错了。假设程序声明了类String(见例13-2)的两个对象,然后在它们之间进行赋值操作。

```
String s1("abc"), s2("def");
s1= s2;
```

这样的操作会出现什么结果呢?对象s1和s2中都只有一个数据成员str,假设在赋值操作前,s1的str指针指向一块动态分配的空间(该空间存放了字符串"abc"),s2的str指针指向另一块动态分配的空间(该空间存放了字符串"def"),赋值语句s1=s2实际执行的操作是s1.str=s2.str。执行该语句之后,对象s1和s2的指针str都指向同一块数据空间(即存

放了"def"的存储空间),另一块存放"abc"的空间就被遗弃了,将无法被访问和释放。此外,如果某个时刻对象 s1 被撤销,其析构函数将释放其指针 str 所指向的数据空间"def"(也就是对象 s2 的指针 str 所指向的数据空间),程序再访问对象 s2 的数据或者撤销对象 s2 时,都会发生指针访问错误。

怎样解决上述问题呢?可以自己定义满足这些特殊需要的赋值运算符重载函数,以覆盖 C++ 提供的默认运算符函数来避免这类错误。例 13-7 重载了运算符 = 和 +,实现了类 String 的对象之间的相加和赋值运算。

【例 13-7】 定义字符串类 String,并重载运算符 + 和 =,使得类 String 的对象可以执行如 x=y+z 这样的操作,表达式 x=y+z 的运算结果是对象 x 中的字符串是对象 y 和 z 中字符串的连接。

```
//文件 string.h: 定义类 String
#if !defined __STRING__H__
#define __STRING__H__

#include <stdlib.h>
#include <iostream.h>

class String
{
    //运算符<<重载函数
    friend ostream &operator<< (ostream &output, const String &s);
    //运算符>>重载函数
    friend istream &operator>> (istream &input, String &s);
public:
    String(char * m="");
    String(String &s);                          //复制构造函数
    ~String();

    //运算符=重载函数
    String & operator= (const String &s);
    //运算符+重载函数
    String operator+ (const String &s) const;

private:
    char * str;
};

#endif

//文件 String.cpp: 类 String 的实现
#include <string.h>
#include "string.h"

#define MAX_STR_LEN 256

String::String(char * m)
{
    str=new char[strlen(m)+1];
```

```
    strcpy(str, m);
}
String::String(String &s)
{
    str=new char[strlen(s.str)+1];
    strcpy(str, s.str);
}
String::~String()
{
    delete []str;
}
//实现运算符<<重载函数
ostream &operator<< (ostream &output, const String &s)
{
    if (s.str !=NULL)
        output<<s.str;
    return output;
}
//实现运算符>>重载函数
istream &operator>> (istream &input, String &s)
{
    char temp[MAX_STR_LEN];
    input>>temp;
    if (s.str !=NULL)
        delete []s.str;
    s.str=new char[strlen(temp)+1];
    strcpy(s.str, temp);
    return input;
}
//实现运算符=重载函数
String &String::operator= (const String &s)
{
    //检查是否自我赋值
    if (&s==this)
        return *this;
    //释放当前对象的数据空间
    delete []str;
    //为当前对象的 str 成员重新分配适当大小的空间
    str=new char[strlen(s.str)+1];
    //将 s 中的字符串复制到当前对象的 str 中
    strcpy(str, s.str);
    return *this;
```

```
}
//实现运算符+重载函数
String String::operator+ (const String &s) const
{
    String res;

    if (res.str != NULL)
        delete []res.str;
    res.str=new char[strlen(str)+strlen(s.str)+1];
    strcpy(res.str, str);
    strcat(res.str, s.str);

    return res;
}
//文件 ex13_7.cpp: 测试重载的运算符
#include <iostream.h>
#include "String.h"

main()
{
    String s1,s2;

    cout<<"Please input two strings:"<<endl;
    cin>>s1>>s2;
    cout<<"s1--"<<s1<<endl<<"s2--"<<s2<<endl;
    s1=s1+s2;
    cout<<"after s1=s1+s2, s1--"<<s1<<endl;

    return 0;
}
```

运行结果：

```
Please input two strings:
Wuhan
Changsha
s1--Wuhan
s2--Changsha
after s1=s1+s2, s1--WuhanChangsha
```

例13-7中的程序为类String重载了运算符=和+,运算符+的重载实现了类String的两个对象相加,其左操作数为类String的对象,所以既可以使用成员函数的方式重载,也可以使用友元函数的方式重载,本程序采用成员函数的方式重载了该运算符。由于两个类String对象相加的结果也是类String的一个对象,所以程序将运算符函数operator+的返回类型处理为String类型。注意operator+返回结果的细节是：函数operator+中运算的结果是局部对象res,执行return res语句时,系统会以res对象为参数自动调用类String的复制构造函数产生一个临时对象(不妨将该对象记为temp);然后,返回main函数的调用点,即语句

```
s1=s1+s2;
```

继续执行赋值运算，即以 temp 为参数调用类 String 的赋值运算重载函数，将结果赋值给 s1。所以，语句 s1 = s1 + s2;的执行过程中先后调用了加运算重载函数、String 的复制构造函数以及赋值运算重载函数。

重载运算符=时，运算符函数第一条语句检测两个操作数是否是同一个对象。如果是则不需进行任何操作，直接返回当前对象本身。如果不是自我赋值，则首先释放当前对象动态申请的空间，然后重新按照参数对象的空间大小分配空间，复制参数对象的数据。运算符函数 operator=在开始的时候检测是否自我赋值非常重要，如果不进行检测，后续语句

```
delete []str;
```

将会释放对象中动态分配的空间，不仅丢失了当前对象的数据，还会使程序在执行语句 strcpy(str, s.str)时出错。

*13.7　类型之间的转换

一个程序经常要处理多种类型的数据，而一个运算一般只操作一种数据类型，如两个整数相加结果仍然是整数。但有些时候一个运算可能需要涉及多种数据类型，如将一个整数和一个浮点数相加，希望的结果是浮点数。对于这类运算，一般在背后都隐含了数据类型的转换操作，如 long 类型的数据和 double 类型的数据相加，要先把 long 类型的数据转换成 double 类型的数据，然后对两个 double 类型的数据执行加操作。计算机系统并没有为任何两种类型数据之间提供足够的运算。因此，在进行运算之前，经常要将一种类型的数据转换成另外一种类型的数据，然后再进行该类型数据的运算。对于内部类型，C++ 知道如何处理这些转换(使用提升规则)，程序员也可通过使用类型强制转换运算符来实现内部类型之间的转换。

如何实现用户自定义类型和其他类型之间的转换呢？系统对自定义类型与内部类型或一种自定义类型与另一种自定义类型之间的关系一无所知，要实现自定义类型和其他类型之间转换，程序员必须明确地告诉编译器如何去进行转换。所以，如何实现自定义类型和其他类型之间的转换，实际上要由程序员自己编程来实现。自定义类型和其他类型之间的转换，就是通过程序员编程来实现如何从一种类型的对象去产生另一种类型的对象。

C++ 提供了两种类型转换方式：一种是通过类型转换运算符函数实现将自定义类的对象转换成其他类型的对象；另外一种是通过转换构造函数实现将其他类型的对象转换成自定义类的对象。

类型转换运算符函数必须是类的非静态成员函数，不能是友元函数。类型转换运算符函数没有参数(如同单目运算符作为成员函数重载)，不能指定返回类型(因为类型转换运算符实际上已经指定了返回类型)。类型转换运算符函数的原型为：

```
operator ClassName () const;
```

其中，ClassName 是要转换的目标类型，可以是内部类型名，也可以是自定义类型名。下面是一个类型转换运算符函数原型的例子：

```
operator int () const;
```

该函数原型声明了一个重载的类型转换运算符函数,它实现将一个自定义类型的对象转换成一个 int 类型的对象。如果 obj 是自定义类的一个对象,则(int)obj 将会产生函数调用 obj.operator int(),该函数将根据调用它的对象 obj 生成一个 int 类型的数据并返回该数据。

例 13-8 在类 String 中声明了一个类型转换运算符函数,实现将类 String 的对象转换成 int 类型。定义好类型转换运算符函数之后,可以明确地通过类型转换运算符调用该函数。更为奇妙的是,当需要的时候,编译器会自动调用这些函数来获得转换结果,就像对内部类型使用提升规则一样。

【例 13-8】 定义字符串类 String,并使用类型转换重载函数实现从类 String 的对象到 int 类型的转换。

分析:类型转换重载函数必须重载为成员函数,而且不能指定返回类型。类型转换运算符是单目运算符,所以其重载函数没有参数。将类 String 的对象转换为 int 类型的重载函数的函数原型为 operator int(),其具体实现采取的是取 str 所指的字符串的长度作为转换结果,当然,也可以根据实际需要确定转换策略,如可以考虑对 str 所指向的字符串进行算术运算获得一个整数等。

```
//文件 string.h: 定义类 String
#if !defined __STRING__H__
#define __STRING__H__

#include <stdlib.h>
#include <iostream.h>

class String
{
    //运算符<<重载函数
    friend ostream &operator<< (ostream &output, const String &s);
    //运算符>>重载函数
    friend istream &operator>> (istream &input, String &s);
public:
    String(char * m="");
    ~String();

    //int 类型转换运算符函数
    operator int () const;

private:
    char * str;
};

#endif

//文件 String.cpp: 类 String 的实现
#include <string.h>
#include "string.h"
```

```
#define MAX_STR_LEN 256
String::String(char * m)
{
    str=new char[strlen(m)+1];
    strcpy(str, m);
}
String::~String()
{
    delete []str;
}
//实现运算符<<重载函数
ostream &operator<< (ostream &output, const String &s)
{
    if (s.str != NULL)
        output<<s.str;
    return output;
}
//实现运算符>>重载函数
istream &operator>> (istream &input, String &s)
{
    char temp[MAX_STR_LEN];
    input>>temp;
    if (s.str != NULL)
        delete []s.str;
    s.str=new char[strlen(temp)+1];
    strcpy(s.str, temp);
    return input;
}
//实现 int类型转换运算符函数
String::operator int() const
{
    return strlen(str);
}
//文件 ex13_8.cpp: 测试类 String
#include <iostream.h>
#include <string.h>
#include "string.h"
int main()
{
    String s;
    cout<<"Please input a string:"<<endl;
```

```
        cin>>s;
        cout<<"Output is:"<<endl;
        cout<<"s as String--"<<s<<endl;
        //将 s转换为 int,并输出转换结果
        cout<<"s as int--"<<(int)s<<endl;
        //将 s自动转换为 int,参与算术运算
        cout<<"10-s="<<10-s<<endl;
        return 0;
    }
```

运行结果:

```
Please input a string:
Changsha
Output is:
s as String--Changsha
s as int--8
10-s=2
```

例13-8中的程序包含3个文件,其中文件string.h是类String的定义。类String的public部分增加了一个类型转换运算符函数:

```
operator int () const;
```

它完成从String类对象到int类型的转换。在文件string.cpp中,函数operator int ()的实现非常简单,只有一条语句:

```
return strlen(str);
```

这意味着任何String对象调用该函数时返回值都是该对象中存放的字符串数据的长度。在实际编程中,可以根据需要选择有意义的转换策略。从这个函数的实现可以看到,类型转换运算符函数的转换过程可以根据需要自行定义。

接下来,主程序对类String进行了测试,首先声明了一个String对象s,读入s的值,运行时输入的值为Changsha。然后程序对类型转换运算符函数的显示转换和隐式转换进行了测试。语句

```
cout<<"s as String--"<<s<<endl;
```

调用流插入运算符重载函数将对象s以String类型的对象输出,这里没有类型的转换。而语句

```
cout<<"s as int--"<<(int)s<<endl;
```

通过表达式(int)s显示调用operator int()函数,转换的结果为整数8,然后将结果输出。最后的语句

```
cout<<"10-s="<<10-s<<endl;
```

是对类型转换运算符函数的隐式调用,由于运算符－不能完成整数和对象 s 之间的减操作,处理表达式 10－s 时,编译器寻找相应的函数:如果有运算符重载函数 operator－的两个参数分别为整型和 String 类型,则调用该函数;如果没有则根据提升规则或寻找类型转换运算符函数,对其中某个参数的类型进行变换,然后再进行同类型数据的运算。这里,编译器会自动调用类 String 的类型转换运算符函数 operator int()将对象 s 转换成整数后参与运算符－的运算,该表达式的运算结果为 2。

类型转换运算符函数实现从自定义类型到其他类型的转换,从其他类型到自定义类型的转换则由转换构造函数实现。转换构造函数也是类的构造函数的一种,其函数原型一般为:

```
ClassName(const OtherClass obj);
```

其中,ClassName 为转换的目标类的名字,OtherClass 为被转换的其他类型的名字,转换构造函数只有一个参数,就是其他类型的对象,转换构造函数将根据该对象初始化自定义类型的对象。同样,转换构造函数的构造过程也是由程序员根据需要定义的。例 13-8 中的构造函数 String(const char * m="")就是一个转换构造函数,它根据一个字符指针类型的对象初始化类 String 的对象。转换构造函数也可以显式或隐式地将一种类型的对象转换成自定义类型的对象以满足运算的需要。

下面例子为类 String 定义了另外一个转换构造函数,并演示了如何隐式地调用转换构造函数。

【例 13-9】 定义字符串类 String,要求类 String 包含两个转换构造函数,分别实现从字符指针类型和 int 类型数据到类 String 对象的转换。

分析:实现从字符串到类对象的转换构造函数的函数原型为 String(const char *),实现从无符号整数到类对象的转换构造函数的函数原型为 String(const int)。这两个构造函数如何实现可以根据需要确定。本例中,用一个字符串初始化字符指针成员 str,首先为 str 分配相应的空间,然后将字符串中的内容复制过来即可。对于用 int 类型的数据来初始化字符指针,为说明转换构造函数的使用,本例使用的方法是将 int 类型的数据转换成字符串,存入 str。当然也可以使用其他方法实现这两个函数。

```
//文件 string.h: 定义类 String
#if !defined __STRING__H__
#define __STRING__H__

#include <stdlib.h>
#include <iostream.h>

class String
{
    //运算符<<重载函数
    friend ostream &operator<< (ostream &output, const String &s);
    //运算符>>重载函数
    friend istream &operator>> (istream &input, String &s);
public:
    //转换构造函数
```

```
    String(const char * m="");
    String(const int len);
    ~String();
    //运算符+=重载函数
    String & operator+=(const String &s);
private:
    char * str;
};
#endif
//文件 String.cpp: 类 String的实现
#include <string.h>
#include <stdio.h>
#include "string.h"
#define MAX_STR_LEN 256
String::String(const char * m)
{
    if (m==NULL)
        str=NULL;
    else
    {
        str=new char[strlen(m)+1];
        strcpy(str, m);
    }
}
String::String(const int len)
{
    char  buffer[MAX_STR_LEN];
    sprintf(buffer, "%d", len);
    str=new char[strlen(buffer)+1];
    strcpy(str, buffer);
}
String::~String()
{
    if (str !=NULL)
        delete []str;
}
//实现运算符<<重载函数
ostream & operator<< (ostream &output, const String &s)
{
    if (s.str !=NULL)
        output<< s.str;
    return output;
```

```
}
//实现运算符>>重载函数
istream & operator>> (istream &input, String &s)
{
    char temp[MAX_STR_LEN];
    input>> temp;
    if (s.str != NULL)
        delete []s.str;
    s.str= new char[strlen(temp)+1];
    strcpy(s.str, temp);
    return input;
}
//实现运算符+=重载函数
String & String::operator+= (const String &s)
{
    char * temp;
    temp= new char[strlen(str)+strlen(s.str)+1];
    strcpy(temp, str);
    strcat(temp, s.str);
    if (str != NULL)
        delete []str;
    str= temp;
    return * this;
}
//文件 ex13_9.cpp: 测试类 String
# include <iostream.h>
# include "String.h"
int main()
{
    String s1, s2;
    cout<< "Please input two strings:"<< endl;
    cin>> s1>> s2;
    cout<< "Output is:"<< endl;
    cout<< "s1--"<< s1<< endl;
    cout<< "s2--"<< s2<< endl;
    //隐式调用转换构造函数
    s1+= "abc";
    cout<< "After s1+=\"abc\"; s1--"<< s1<< endl;
    s2+= 10;
    cout<< "After s2+=10; s2--"<< s2<< endl;
    //显式调用转换构造函数
    cout<< "(String)15--"<< (String)15<< endl;
```

```
        cout<<"(String)\"abc\"--"<<(String)"abc"<<endl;
        return 0;
}
```

运行结果：

```
Please input two strings:
Wuhan
Changsha
Output is:
s1--Wuhan
s2--Changsha
After s1+="abc"; s1--Wuhanabc
After s2+=10; s2--Changsha10
(String)15--15
(String)"abc"--abc
```

例13-9中类String的定义包含两个转换构造函数：

```
String(const char * m="");
String(const int len);
```

第一个构造函数根据字符指针指向的字符串初始化对象，实现方法是为str分配能容纳m指向的字符串的空间，然后将该字符串复制过来。第二个构造函数根据一个整数来初始化对象，其实现方法则是通过sprintf函数将int类型的数据转换到字符数组中，然后为str分配空间，并将转换得到的字符串复制到该空间。

main函数首先声明了类String的两个对象s1和s2，然后调用流提取运算符重载函数输入这两个对象，程序的输入分别为Wuhan和Changsha。接下来程序输出了对象s1和s2，并对转换构造函数进行了测试。语句

```
s1+="abc";
```

调用了类String的运算符重载函数operator+=，但函数operator+=要求的参数类型是类String的对象，和字符串"abc"的类型不符合。这种情况下，程序会自动调用转换构造函数String(const char * m="")，以字符串"abc"为参数创建类String的一个临时对象参与运算。运算符重载函数operator+=实际上是将对象s1和这个临时对象相加。语句

```
s2+=10;
```

也调用了类String的运算符重载函数operator+=，参数类型也不相符。程序自动调用转换构造函数String(const int len)，以整数10为参数创建类String的一个临时对象参与运算。临时对象中的字符串为"10"，所以相加后对象s2中str的值为"Changsha10"。这两条语句是对转换构造函数的隐式调用。语句

```
cout<<"(String)15--"<<(String)15<<endl;
```

则是对转换构造函数的显式调用。表达式(String)15将整数15转换成String类型的对象，程序将以15为参数调用类String的第二个转换构造函数创建一个临时对象。语句

```
cout<<"(String)\"abc\"--"<<(String)"abc"<<endl;
```

将字符串"abc"转换成 String 类型的对象，程序以"abc"为参数调用类 String 的第一个转换构造函数创建一个临时对象，然后通过重载 <<运算符将该对象输出。

习　题　13

13.1　填空：

(1) 当用成员函数重载双目运算符时，运算符的左操作数必定是________。

(2) 单目运算符，作为友元函数重载时有________个形参。

(3) 在一个类中可以对一个操作符进行________种重载。

(4) 双目运算符重载为成员函数时带________个参数，重载为友元函数时带________个参数。

(5) 运算符的重载函数________改变运算符的优先级。

13.2　运算符重载函数可以使用默认参数吗？如果可以，将会产生什么后果？

13.3　定义包含年、月、日信息的日期类 Date，并重载二元运算符+，使之具有日期对象和整数(天数)相加的功能，并编程测试。

13.4　为例 13-1 中的类 Complex 增加运算符 * 的重载，实现复数对象的乘法运算，并编写测试程序。

13.5　为习题 13.4 中的类 Complex 增加前自增、前自减、后自增、后自减运算符的重载，并编写测试程序。注意：自增和自减都只对复数对象的实部作加减 1 运算。

13.6　对于下面的类 MyString，要求重载一些运算符后可以计算表达式：a=b+c;，其中 a、b、c 都是类 MyString 的对象。请重载相应的运算符并编写程序测试。

```
class MyString {
public:
    MyString(char * s) {
        str=new char[strlen(s)+1];
        strcpy(str, s);
    }
    ~MyString() {
        delete []str;
    }
private:
    char * str;
};
```

13.7　对于下面的 Time 类：

```
class Time {
public:
Time(int h=0, int m=0, int s=0);
private:
    int hour, minute, second;
};
```

要求：

(1) 实现构造函数，是类 Time 的数据成员可以得到正确的赋值。

(2) 重载流插入运算符(<<),按照 hh:mm:ss 的格式输出类 Time 的对象。

(3) 重载流提取运算符(>>),按照 hh-mm-ss 的格式输入类 Time 的对象。

13.8 下面是数组类 Array 的定义:

```
class Array {
public:
Array(int n);
    ~Array();
    int & operator [](int n);
    Array & operator= (Array &obj);
    int operator== (Array &obj);
private:
    int size;
    int * ptr;
};
```

要求:

(1) 实现构造函数,为类 Array 的对象分配一个包含 n 个数组元素的整型数组。

(2) 实现析构函数,做好撤销对象前的清理工作。

(3) 实现运算符[]的重载函数,如果 a 是类 Array 的对象,y 是整型变量,则 a[n]=y;可以为对象 a 的数组的第 n 个元素赋值,函数的实现要防止数组越界。

(4) 实现运算符=的重载函数,使类 Array 的对象之间能正确赋值。

(5) 实现运算符==的重载函数,使之可以判断两个数组是否相等。

(6) 编写程序测试类 Array。

13.9 用运算符重载的方法实现习题 11.10 的 HugeInt 类的所有功能,并为类 HugeInt 重载前自增、前自减、后自增、后自减 4 个运算符,使之能实现正确的数字运算,然后编写程序验证。

13.10 根据例 13-8 和例 13-9,改写 String 类的类型转换函数和转换构造函数,要求实现下面的转换功能。

(1) String 对象到 int 类型的转换:如果 String 对象中数据是整数的字符串形式,则将该整数作为将该 String 对象转换到 int 的结果,否则转换成整数 0。

(2) float 类型到 String 对象的转换:将给定的浮点数转换成字符串形式,并存入生成的 String 对象。

(3) String 对象到 float 类型的转换:如果 String 对象中数据是浮点数的字符串形式,则将该浮点数作为将该 String 对象转换到 float 的结果,否则转换成浮点数 0。

编写程序验证上述功能。

第 14 章 继 承

【学习内容】

本章将介绍 C++ 的继承机制。主要内容包括：

◆ 继承和派生的基本概念。

◆ 继承的定义。

◆ 在派生类中重定义基类的函数。

◆ 基类和派生类对象的转换。

◆ 类指针的使用。

◆ 派生类中构造函数的定义及继承关系中的构造函数之间、析构函数之间的关系。

◆ 多重继承。

◆ 复合与继承的关系。

【学习目标】

◆ 理解继承的基本概念。

◆ 掌握如何通过继承建立类的层次结构。

◆ 掌握通过类指针操作继承关系中对象的方法。

◆ 掌握如何使用复合和继承进行软件的渐增式开发。

◆ 了解多重继承。

继承是面向对象方法的基本特征之一，是软件重用的一种重要的形式。通过继承机制，可以在已有的类的基础上建立新类，新类可以继承已有类的属性和行为，也可以修改已有类的属性和行为，或者增加新的属性和行为以满足自身特殊的需要。继承缩短了定义新类的时间，并可以重用已经经过调试和测试的高质量的代码(包括自己或别人的代码)，减少最终系统出错的可能性。继承是面向对象程序设计的一个非常重要的概念，也是处理复杂软件的一个非常有效的技术。

本章介绍继承的概念，以及 C++ 继承的定义和使用方法。

14.1 继承和派生的概念

类是对现实生活中实体的共性的抽象描述。现实世界中，实体之间普遍存在着两种关系，即“是(is_a)”和“拥有(has_a)”的关系。第 12 章介绍的类的复合描述的是“拥有(has_a)”的

关系,继承描述的则是实体之间的"是(is_a)"的关系。

看一个具体的例子。假设要建立一个学校信息管理系统,要求能管理大、中、小学等各类学校的基本信息,并能为外界提供查询和检索功能。首先分析系统需求,大学的基本信息应该包括校名、校址、简单校史、是否是重点大学、学院信息、师资力量、招生人数、在读学生数量等,中学基本信息包括校名、校址、属于哪一级重点中学、在读学生数量、升学率等,小学信息包括校名、校址、是否是重点小学、在读学生数量、升学率、开设课程等。由于大学、中学和小学需要描述的信息不完全相同,因此需要分别为它们建立不同的类(如类 College、类 HighSchool 和类 ElementarySchool)。这样当然可以描述所有学校,但是有两点不尽如人意。首先,3 个类所描述的信息有一部分相同,都包含校名、校址等,这部分信息没有被重用,而是在不同的类中分别进行了重复定义,和它们相关的一些功能相同的函数(如查询学校校名等)也必须分别在 3 个不同的类中给出几乎完全一样的定义和实现。其次,所有的大学作为类 College 的实例可以进行统一的管理和操作,所有的中学作为类 HighSchool 的实例也可以进行统一的管理和操作,但大学、中学、小学作为不同类的对象却不可能作为"学校"这一类进行统一的操作和管理,这和现实生活中的实际情况不符,同时也会增加程序设计的复杂性。

如果使用继承,这些问题就都可以迎刃而解。继承允许以现有的类为基础来构建新的类,新的类继承了现有类的属性和行为,并且可以根据自己的需要修改这些属性和行为,也可以增加新的属性和行为。继承可以使程序员重用已经定义的、经过调试和测试的高质量的代码,提高程序质量。在类的继承体系中,被继承的类称为基类,新定义的类称为派生类。在继承基类属性和行为的同时,派生类可以添加自己的数据成员和函数成员。继承把派生类和基类联系了起来,派生类对象同时也可以被认为是基类的对象,这样程序员可以统一管理和操作所有派生类和基类的对象。

对于上述问题,可先建立一个学校类(基类 School),在该类中定义 3 个类的共同部分(校名、校址等共有的属性和相关的操作),然后分别建立 3 个派生类(类 College、类 HighSchool 和类 ElementarySchool)并让它们继承类 School,在 3 个派生类中就只需要定义类 School 中没有的、属于各自特有的数据成员和函数成员即可。这样不仅实现了代码重用,而且由于 3 个派生类拥有共同的基类 School,它们的对象也都可被看做是基类 School 的对象,因此可对它们进行统一操作和管理。

由此可见,继承机制为描述客观世界的层次关系提供了直观、自然和方便的描述手段,定义的新类可以直接继承类库中定义的或其他人定义的高质量的类,而新的类又可以成为设计其他类的基础,这样软件重用就变得更加方便、自然。

14.2 继承的定义

上一节介绍了继承的作用及意义,本节将讨论如何定义基类和派生类、派生类的继承方式以及派生类和基类对象之间的关系等。

14.2.1 派生类和基类

现实世界中的事物之间的关系存在着从属、一般和特殊的关系,一个类的对象经常也可

以视为另一个类的对象。例如，狮子 A 是狮子类的对象，同时它也是哺乳类动物的对象，因为所有狮子都属于胎生动物类，其他动物如老虎、猴子等也都是这样。可以说狮子类 Lion 是从胎生动物类 Viviparity 继承而来的，Viviparity 是基类，Lion 是派生类。所有的派生类对象都是基类对象，但不能说所有基类对象都是派生类对象。在这个例子中，所有的狮子都是胎生动物类动物，但不是所有胎生动物类动物都是狮子。

继承使基类和派生类之间具有层次关系，并形成了类的树形结构。一个类可以单独存在，既不继承于其他类，也不被其他类继承。但一旦使用继承机制定义一个类时，它就成为树形结构中的一个结点，它既可以作为基类被其他类继承，为派生类提供共同的属性和行为，也可以作为派生类从其他的类继承它们的属性和行为。

下面是描述脊索动物类 Chordata 的一个继承层次结构。脊索动物 Chordata 可分为鱼类 Fish、两栖类 Amphibiotic、爬虫类 Reptilia、鸟类 Aves 和哺乳类 Mammal，鱼类 Fish 又可分为软骨鱼 CartilaginousSkeleton 和硬骨鱼 BonySkeleton 两个子类，哺乳类动物还可以划分为卵生动物 Ovoviviparous、有袋动物 Marsupial 和胎生动物 Viviparity 等子类，人 Human 属于胎生动物。图 14-1 描述了这种通过继承而形成的层次结构。

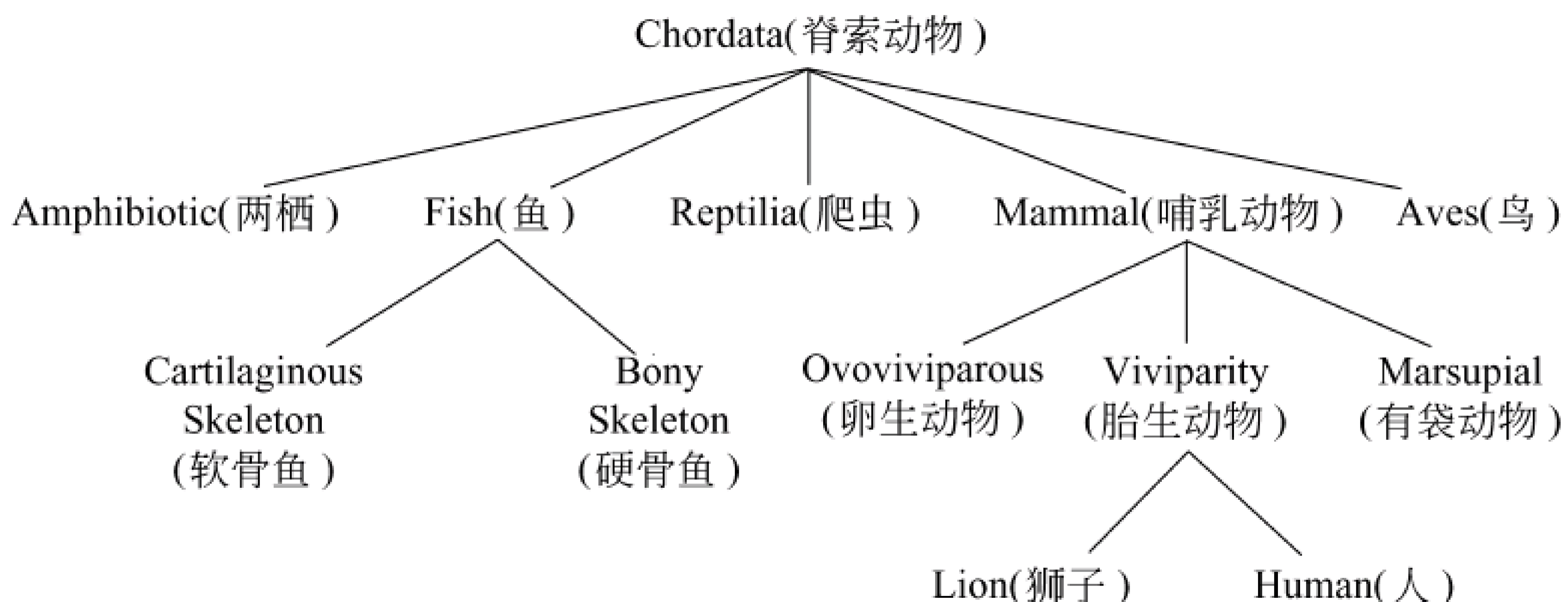


图 14-1　脊索动物的继承层次结构

这种继承结构的例子很多，如英语单词 Word 类可以分为动词 Verb 类、名词 Noun 类、形容词 Adjective 类等，动词又可分为及物动词 TransitiveVerb 类、不及物动词 InTransitiveVerb 类，名词还可分为可数名词 Noun_of_discontinuous_quantity 类和不可数名词 Noun_of_continuous_quantity 类等。

通过继承定义派生类时，需要明确指明它是从哪个基类派生而来。例 14-1 中的程序首先定义了基类 Viviparity，然后定义了派生类 Lion 和 Human。

【例 14-1】　继承示例。

```
class Viviparity{
public:
    Viviparity ();
    ~Viviparity ();
    …
};
class Lion: public Viviparity {
public:
    Lion();
```

```
        ~Lion();
        …
    };
    class Human: public Viviparity {
    public:
        Human();
        ~Human();
        …
    };
```

类Lion和Human都是类Viviparity的派生类。在类Lion的定义中可以看到,类名Lion的后面由冒号引出了关键字public和基类名Viviparity,这种继承是公有继承,也是最常用的一种继承方式。在公有继承中,基类中的公有成员被派生类作为公有成员继承,基类中受保护成员被派生类作为受保护成员继承。

继承可以节省代码,可以将各个派生类中共有的部分抽取出来放到基类中去。但更重要的是,由于派生类继承了基类的所有属性,C++允许将派生类的对象作为基类对象处理,这使得编写更灵活的程序成为可能。

14.2.2 继承的方式

C++有3个访问控制说明符对类的成员进行访问控制:public、private和protected。具有public属性的成员对外界是可见的,可以直接访问;具有private属性的成员对外界是不可见的,只能被该类的成员函数访问;具有protected属性的成员对外界也是不可见的,也不能被外界函数直接访问。protected成员和private成员是否一样呢?孤立地研究一个类,二者没有什么差别,它们的差别在继承时才能体现出来,基类的protected成员可以被派生类的成员函数访问,而基类的private成员则不能被派生类的成员函数访问。

从基类派生出新的类时,类的继承也有3种方式:公有继承(public)、受保护继承(protected)和私有继承(private)。3种继承方式的区别体现在基类中的成员被派生类继承后成员对外的可见性会有所不同。使用公有继承时,基类中的公有成员被继承后在派生类中仍然是公有的,基类中受保护的成员被继承后在派生类中仍然是受保护的。使用受保护继承时,基类中的公有成员和受保护成员被继承后在派生类中都是受保护的。使用私有继承时,基类中的公有成员和受保护成员被继承后在派生类中都是私有的。

对于基类中的私有成员,不管使用哪种继承方式,派生类中的(不是从基类中继承来的)成员函数都不能直接访问,而只能通过定义于基类中的其他公有成员函数或受保护成员函数去访问。但是,这并不是说基类中的私有属性不被继承,声明一个派生类的对象时,该对象不仅拥有派生类中定义的所有属性,还拥有定义在基类中的所有属性,只是对定义于基类中的私有属性的访问方式进行了限制。

定义受保护继承或私有继承的语法和定义公有继承的语法完全一样,只是在派生类的定义中所使用的用于声明继承方式的关键字不同。例14-2中定义的派生类Lion继承基类Viviparity时使用的是私有继承方式。

【例14-2】 私有继承示例。

```
class Viviparity {
public:
    Viviparity ();
    ~Viviparity ();
    ...
};
class Lion: private Viviparity {
public:
    Lion();
    ~Lion();
    ...
};
```

14.2.3 类的层次

一个类可以是某一个继承关系中的基类，同时也可以是另一个继承关系中的派生类。如果类 A 派生出类 B，则类 A 是类 B 的直接基类。如果类 B 又派生出类 C，则类 B 是类 C 的直接基类，而类 A 是类 C 的间接基类。

定义派生类时，要在派生类的头部明确列出其直接基类，间接基类则不用列出。派生类不仅继承了直接基类的成员，同时也继承了所有间接基类的成员。

下面例子定义了具有继承关系的 A、B、C 3 个类。

【例 14-3】 间接基类示例。

```
class A {
protected:
    int aMember;
    ...
};
class B: protected A {
protected:
    int bMember;
    ...
};
class C: private B {
protected:
    int cMember;
    ...
};
```

类 A 拥有受保护数据成员 aMember，类 B 以受保护方式继承了类 A。类 B 就拥有两个数据成员：受保护的成员 bMember 和 aMember。其中，aMember 是在基类 A 中定义、类 B 通过的受保护继承而获得的；类 C 采取私有方式继承了类 B，因而类 C 拥有 3 个成员：受保护数据成员 cMember、私有数据成员 bMember 和 aMember，分别从直接基类 B 和间接基类 A 中继承而来。

14.2.4 在派生类中重定义基类的函数

派生类自动继承了基类中定义的数据成员和成员函数。如果派生类认为基类中某个成员函数的功能不能满足需要,可以在派生类中重新定义该函数。重定义(overriding)基类的成员函数需要使用和该函数相同的函数名和参数列表,如果参数列表不同,就是函数重载而不是函数的重定义了。经过重定义之后,派生类定义的函数取代了基类中原来的函数定义。也就是说,通过派生类对象访问到的该成员函数是在派生类中重新定义的版本,基类中定义的版本不能通过派生类对象访问。如果派生类的成员函数需要访问基类中被重定义的函数,可以使用基类名加作用域运算符(::)的方式。

重新考虑例10-1中的类Media,为了表示音乐媒体和影视媒体,类Media引入了数据成员type,通过给type设置不同的值来区分不同类型的媒体。这种做法存在一些缺陷,就是不同类型的媒体描述的信息要求完全相同(因为它们都是同一个类的实例化对象),而实际上这些媒体在内容的表示上是有差别的。例如,音频媒体需要描述专辑名、歌手、制作人、制作公司等,而视频媒体则需要描述作品名、导演、演员、制作公司等。为了更好地描述这些不同的信息,可以使用继承机制,创建包含公共信息的基类Media,然后从类Media派生出音频媒体类AudioMedia和视频媒体类VideoMedia,在类Media中有一个输出媒体基本信息的成员函数showInfo,这个函数将被类AudioMedia和类VideoMedia继承。但很显然,对于两个派生类来说,输出的这些信息还不够详细,因此需要在派生类中重定义该函数。

【例14-4】 采用继承机制重新定义例10-1中的Media类,将音乐媒体和影视媒体类分开,并根据需要在各派生类中重定义基类中的函数。

```
//文件 media.h: 定义类 Media
#if !defined __MEDIA__H__
#define __MEDIA__H__

class Media
{
public:
    //构造函数
    Media(char * n,char * c);
    //析构函数
    ~Media();
    void showInfo();

private:
    char * name;                    //媒体名
    char * company;                 //制作公司
};

#endif

//文件 media.cpp: 类 Media 的实现
#include <string.h>
#include <iostream.h>
#include "media.h"
```

```
//构造函数的实现
Media::Media(char * n,char * c)
{
    name=new char[strlen(n)+1];
    strcpy(name,n);
    company=new char[strlen(c)+1];
    strcpy(company,c);
}
//析构函数的实现
Media::~Media()
{
    if (name!=NULL)
        delete []name;
    if (company!=NULL)
        delete []company;
}
void Media::showInfo()
{
    cout<<"Name: "<<name<<endl;
    cout<<"Company: "<<company<<endl;
}
//文件 audioMedia.h:定义类 AudioMedia
#if !defined __AUDIOMEDIA_H__
#define __AUDIOMEDIA_H__
#include "media.h"
class AudioMedia: public Media
{
public:
    AudioMedia(char * n,char * c,char * s);
    ~AudioMedia();
    void showInfo();
private:
    char * singer;                                    //歌手
};
#endif
//文件 audioMedia.cpp:类 AudioMedia 的实现
#include <string.h>
#include <iostream.h>
#include "audioMedia.h"
AudioMedia::AudioMedia(char * n,char * c,char * s)
:Media(n,c)                                           //调用基类的构造函数
{
    singer=new char[strlen(s)+1];
```

```
        strcpy(singer,s);
    }
    AudioMedia::~AudioMedia()
    {
        if (singer!=NULL)
            delete []singer;
    }
    void AudioMedia::showInfo()
    {
        Media::showInfo();                              //调用基类被重定义的函数
        cout<<"Singer: "<<singer<<endl;
    }
    //文件 videoMedia.h: 定义类 VideoMedia
    #if !defined __VIDEOMEDIA__H__
    #define __VIDEOMEDIA__H__
    #include "media.h"
    class VideoMedia: public Media
    {
    public:
        VideoMedia(char * n,char * c,char * d,char * a);
        ~VideoMedia();
        void showInfo();
    private:
        char * director;                                //导演
        char * actor;                                   //演员
    };
    #endif
    //文件 videoMedia.cpp: 类 VideoMedia 的实现
    #include <string.h>
    #include <iostream.h>
    #include "videoMedia.h"
    VideoMedia::VideoMedia(char * n,char * c,char * d,char * a)
    : Media(n,c)                                        //调用基类的构造函数
    {
        director=new char[strlen(d)+1];
        strcpy(director,d);
        actor=new char[strlen(a)+1];
        strcpy(actor,a);
    }
    VideoMedia::~VideoMedia()
    {
        if (director!=NULL)
```

```
        delete []director;
    if (actor!=NULL)
        delete []actor;
}
void VideoMedia::showInfo()
{
    Media::showInfo();                              //调用基类被重定义的函数
    cout<<"Director: "<<director<<endl;
    cout<<"Actor: "<<actor<<endl;
}
//文件 ex14-4.cpp: 测试类 AudioMedia 和类 VideoMedia
#include <iostream.h>
#include "audiomedia.h"
#include "videomedia.h"
main()
{
    AudioMedia audio("The Colour of My Love","Columbia","Celine Dion");
    VideoMedia video("Forrest Gump","Paramount Pictures","Robert Zemeckis","Tom Hanks");
    cout<<"Audio Media:"<<endl;
    audio.showInfo();
    cout<<endl;
    cout<<"Video Media:"<<endl;
    video.showInfo();
    return 0;
}
```

运行结果：

```
Audio Media:
Name: The Colour of My Love
Company: Columbia
Singer: Celine Dion
Video Media:
Name: Forrest Gump
Company: Paramount Pictures
Director: Robert Zemeckis
Actor: Tom Hanks
```

例 14-4 中，基类 Media 的定义由两个私有数据成员(name 和 company)和 3 个成员函数(构造函数、析构函数和 showInfo 函数)组成。构造函数接收两个参数，分别用来初始化两个私有数据成员，并为两个私有数据成员分配存储相应数据的空间。析构函数释放动态分配的内存空间，防止内存泄漏。函数 showInfo 的实现非常简单，直接输出对象的 name 和 company 信息。

派生类 AudioMedia 是从类 Media 按照公有方式继承而来的。类 AudioMedia 的定义只包含一个私有数据成员(singer)和3个成员函数(构造函数、析构函数和 showInfo 函数),注意这里的 showInfo 函数的函数原型和基类 Media 中的成员函数 showInfo 的函数原型完全相同,是对基类函数的重定义。这样类 AudioMedia 就拥有两个 showInfo 函数(一个是自己定义的,另一个是从基类继承而来的),并且都可以访问到(以不同的方式)。

类 AudioMedia 的构造函数接收3个参数,其中参数 n 和 c 是用来初始化定义于基类中的数据成员 name 和 company 的,类 AudioMedia 的构造函数显式地调用基类 Media 的构造函数,并将相应的参数传递过去。显式地调用基类构造函数的格式为:

```
AudioMedia::AudioMedia(char * n,char * c,char * s)
:Media(n,c)                    //调用基类的构造函数
{…}
```

在构造函数的参数列表之后用冒号引出基类构造函数的函数名和参数。类 AudioMedia 的构造函数的另外一个参数(s)用来初始化派生类自己定义的数据成员 singer。

类 AudioMedia 的析构函数释放了动态分配的空间,防止内存泄漏。

派生类 VideoMedia 的定义和实现和类 AudioMedia 类似,只是在描述的信息有所区别。类 VideoMedia 描述了导演(director)和演员(actor)的信息,而没有歌手(singer)的信息。

派生类不能继承基类的构造函数和析构函数,派生类的构造函数和析构函数在运行时会自动调用基类的构造函数和析构函数。实例化派生类的对象时,程序会自动调用派生类的构造函数,而派生类的构造函数在执行函数体之前会先调用基类的构造函数,所以派生类的构造函数只需要初始化在派生类中定义的数据成员,从基类继承的数据成员则交给基类的构造函数去初始化。

派生类的构造函数调用基类的构造函数的方式有两种:一种是例14-4中的显式调用的方式,在参数列表之后用冒号引出基类构造函数的函数名和参数,如果需要给基类的构造函数传递参数时,必须使用显式调用的方式;另一种是隐式调用的方式,如果要调用的是基类的默认构造函数(不需要传递参数),则可以使用隐式调用,采用隐式调用时派生类构造函数的实现中不需要列出基类构造函数的函数名,系统会自动执行基类的默认构造函数来完成基类中数据的初始化。

派生类的析构函数也会自动调用基类的析构函数,具体时机是在执行完派生类的析构函数之后。所以,派生类的析构函数也只需要释放自己的数据成员维护的动态分配的空间,基类的数据成员维护的动态分配的空间则交给基类的析构函数去释放。

类 AudioMedia 重定义了基类 Media 的成员函数 showInfo,希望除了输出在基类中定义的基本信息外,还要输出作为一个音乐媒体的特定相关信息。由于媒体名字(name)是定义于基类中的私有数据成员,类 AudioMedia 的成员函数 showInfo 无法直接访问,只能调用基类的 showInfo 函数帮助输出这部分信息,然后再输出歌手信息。派生类的成员函数调用基类中被重定义的成员函数要使用基类名加作用域运算符,在类 AudioMedia 的成员函数 showInfo 中调用基类 Media 的成员函数 showInfo 的方法为:

```
Media::showInfo();
```

如果不加上基类名和作用域运算符，程序调用的将是派生类自己的 showInfo 函数，将会导致无穷的递归调用。

14.2.5 派生类和基类的转换

派生类对象也是基类对象，因此派生类对象可以作为基类对象来处理。但实际上二者的类型是不同的，派生类对象具有派生类类型，而基类对象具有基类类型。

把派生类对象作为基类对象处理是合理的，因为派生类继承了基类中定义的所有属性和行为。因此，把派生类对象作为基类对象使用时，访问该对象中定义于基类中的任何属性都不会出错。同样的道理，把派生类对象直接赋值给基类对象也是合法的。如类 Derived 从类 Base 继承而来，下面的语句是合法的。

```
Derived bobj;
Base aobj=bobj;
```

但是，反过来把基类对象作为派生类对象处理就不行了。因为派生类中还可能定义了基类中没有的成员，将基类对象作为派生类对象使用时，如果访问到对象中定义于派生类中的成员时就会出错。同样，把基类对象直接赋值给派生类对象也是非法的，除非提供了正确的转换构造函数或者正确地重载了赋值运算符。

14.3 类 指 针

定义一个类实际上就是定义一种新的用户自定义类型。可以像使用系统预定义类型那样来使用自定义类型，如可以用类名来声明变量(类的对象)、数组、指针等。通过类的指针，可以灵活地操作类的对象。

引入继承的概念之后，公有派生类的对象可以无条件地作为基类的对象处理，这使得通过指针去操作对象变得更加灵活和方便。例如，从一个基类可以派生出多个派生类(例 14-4 中从基类 Media 派生出了两个派生类)，尽管这些派生类的对象的类型各不相同，但由于它们都可以看做是基类的对象，依然可以建立由这些不同的对象构成的数组或链表，并通过基类类型指针来统一操作它们。反之则不行，基类的对象不能直接当作派生类的对象来处理。

用指针操作一个对象时，除了可以定位一个对象外，还可能包含隐式的类型转换，即不管指针实际指向的对象是什么类型，它都会把该对象当作该指针类型的对象。如果通过指向基类的指针来操作派生类对象，派生类对象就会被当作基类对象处理；反之，如果通过指向派生类的指针来操作基类对象，则基类对象也会被当作派生类对象使用，当然这样使用存在潜在的危险。使用派生类指针操作基类对象时需要进行类型的强制转换，并且不能访问定义于派生类中的属性，否则可能会出现不可预测的错误。

例 14-5 修改了例 14-4 中的 3 个类的定义。在基类(类 Media)和派生类(类 AudioMedia 和类 VideoMedia)中定义了访问私有数据成员的成员函数，并试图用基类指针(Media * 类型)和派生类指针(AudioMedia * 类型和 VideoMedia * 类型)分别操作基类和

派生类的对象。

【例14-5】 修改例14-4中定义的3个类,并尝试分别使用基类和派生类的指针操作基类和派生类的对象。

```
//文件 media.h: 定义类 Media
#if !defined __MEDIA__H__
#define __MEDIA__H__

class Media
{
public:
    //构造函数
    Media(char * n,char * c);
    //析构函数
    ~Media();

    char * getName(char * str);
    char * getCompany(char * str);

private:
    char * name;                    //媒体名
    char * company;                 //制作公司
};

#endif

//文件 media.cpp: 类 Media 的实现
#include <string.h>
#include <iostream.h>
#include "media.h"

//构造函数的实现
Media::Media(char * n,char * c)
{
    name=new char[strlen(n)+1];
    strcpy(name,n);
    company=new char[strlen(c)+1];
    strcpy(company,c);
}

//析构函数的实现
Media::~Media()
{
    if (name!=NULL)
        delete []name;
    if (company!=NULL)
        delete []company;
}

char * Media::getName(char * str)
{
```

```
    strcpy(str,name);
    return str;
}
char * Media::getCompany(char * str)
{
    strcpy(str,company);
      return str;
}
//文件 audioMedia.h: 定义类 AudioMedia
#if !defined __AUDIOMEDIA__H__
#define __AUDIOMEDIA__H__
#include "media.h"
class AudioMedia: public Media
{
  public:
    AudioMedia(char * n,char * c,char * s);
    ~AudioMedia();
    char * getSinger(char * str);
  private:
    char * singer;                                   //歌手
};
#endif
//文件 audioMedia.cpp: 类 AudioMedia 的实现
#include <string.h>
#include <iostream.h>
#include "audioMedia.h"
AudioMedia::AudioMedia(char * n,char * c,char * s)
: Media(n,c)                                         //调用基类的构造函数
{
    singer=new char[strlen(s)+1];
    strcpy(singer,s);
}
AudioMedia::~AudioMedia()
{
    if (singer!=NULL)
        delete []singer;
}
char * AudioMedia::getSinger(char * str)
{
    strcpy(str,singer);
    return str;
}
```

```
//文件 videoMedia.h: 定义类 VideoMedia
#if !defined __VIDEOMEDIA__H__
#define __VIDEOMEDIA__H__
#include "media.h"

class VideoMedia: public Media
{
  public:
    VideoMedia(char * n,char * c,char * d,char * a);
    ~VideoMedia();

    char * getDirector(char * str);
    char * getActor(char * str);

  private:
    char * director;                                   //导演
    char * actor;                                      //演员
};

#endif

//文件 videoMedia.cpp: 类 VideoMedia 的实现
#include <string.h>
#include <iostream.h>
#include "videoMedia.h"

VideoMedia::VideoMedia(char * n,char * c,char * d,char * a): Media(n,c)
//调用基类的构造函数
{
    director=new char[strlen(d)+1];
    strcpy(director,d);
    actor=new char[strlen(a)+1];
    strcpy(actor,a);
}

VideoMedia::~VideoMedia()
{
    if (director!=NULL)
        delete []director;
    if (actor!=NULL)
        delete []actor;
}

char * VideoMedia::getDirector(char * str)
{
    strcpy(str,director);
    return str;
}

char * VideoMedia::getActor(char * str)
{
```

```
    strcpy(str,actor);
      return str;
}
//文件 ex14_5.cpp:类指针的使用
#include <iostream.h>
#include "audioMedia.h"
#include "videoMedia.h"

#define MAX_STR_LEN 256

int main()
{
    Media medium("Slumdog Millionaire","Celador Films");
    AudioMedia audio("The Colour of My Love","Columbia","Celine Dion");
    VideoMedia video("Forrest Gump","Paramount Pictures","Robert Zemeckis",
    "Tom Hanks");

    char strBuffer[MAX_STR_LEN];
    //用基类指针操作对象
    Media * mPtr;
    //用基类指针指向基类对象
    mPtr=&medium;
    cout<<"Accessing Media object through Media * "<<endl;
    cout<<"Name: "<<mPtr->getName(strBuffer)<<endl;
    cout<<"Company: "<<mPtr->getCompany(strBuffer)<<endl;

    cout<<endl;
    //用基类指针指向派生类对象
    mPtr=&audio;
    cout<<"Accessing AudioMedia object through Media * "<<endl;
    cout<<"Name: "<<mPtr->getName(strBuffer)<<endl;
    cout<<"Company: "<<mPtr->getCompany(strBuffer)<<endl;
    //要调用定义于派生类中的函数必须进行类型的强制转换
    cout<<"Singer: "<< ((AudioMedia * )mPtr)->getSinger(strBuffer)<<endl;

    cout<<endl;
    //用基类指针指向派生类对象
    mPtr=&video;
    cout<<"Accessing VideoMedia object through Media * "<<endl;
    cout<<"Name: "<<mPtr->getName(strBuffer)<<endl;
    cout<<"Company: "<<mPtr->getCompany(strBuffer)<<endl;
    //要调用定义于派生类中的函数必须进行类型的强制转换
    cout<<"Director: "<< ((VideoMedia * )mPtr)->getDirector(strBuffer)<<endl;
    cout<<"Actor: "<< ((VideoMedia * )mPtr)->getActor(strBuffer)<<endl;

    cout<<endl;
    //用派生类指针操作对象
```

```
    AudioMedia * amPtr;
    //用派生类指针指向基类对象
    amPtr= (AudioMedia * ) &medium;
    cout<< "Accessing Media object through AudioMedia * "<< endl;
    cout<< "Name: "<< amPtr- > getName(strBuffer)<< endl;
    cout<< "Company: "<< amPtr- > getCompany(strBuffer)<< endl;
    //危险,访问了不存在的属性
    cout<< "Singer: "<< amPtr- > getSinger(strBuffer)<< endl;

    cout<< endl;

    //用派生类指针指向派生类对象
    amPtr= &audio;
    //调用基类中定义的函数
    cout<< "Accessing AudioMedia object through AudioMedia * "<< endl;
    cout<< "Name: "<< amPtr- > getName(strBuffer)<< endl;
    cout<< "Company: "<< amPtr- > getCompany(strBuffer)<< endl;
    cout<< "Singer: "<< amPtr- > getSinger(strBuffer)<< endl;

    return 0;
}
```

运行结果:

```
Accessing Media object through Media *
Name: Slumdog Millionaire
Company: Celador Films

Accessing AudioMedia object through Media *
Name: The Colour of My Love
Company: Columbia
Singer: Celine Dion

Accessing VideoMedia object through Media *
Name: Forrest Gump
Company: Paramount Pictures
Director: Robert Zemeckis
Actor: Tom Hanks

Accessing Media object through AudioMedia *
Name: Slumdog Millionaire
Company: Celador Films
Singer: ? ↕

Accessing AudioMedia object through AudioMedia *
Name: The Colour of My Love
Company: Columbia
Singer: Celine Dion
```

例 14-5 中的程序定义了基类 Media、派生类 AudioMedia 和 VideoMedia。首先分析类 Media 的定义,除了构造函数和析构函数外,还有两个成员函数 getName 和 getCompany,

分别用来访问私有数据成员 name 和 company。类 AudioMedia 和类 VideoMedia 继承了类 Media,类 AudioMedia 的构造函数有 3 个参数:字符指针 n 和 c 用来初始化在基类 Media 中定义的数据成员 name 和 company,另外一个参数 s 则用来初始化在派生类 AudioMedia 中定义的数据成员 singer。所以,这里需要将部分参数传递给基类的构造函数,必须显式地调用基类的构造函数,相应地语句为:

```
AudioMedia::AudioMedia(char * n,char * c,char * s)
:Media(n,c)                    //调用基类的构造函数
```

派生类 VideoMedia 的构造函数也有类似的处理。

在 main 函数中,程序中声明了类 Media 的对象 medium 和指针 mPtr、类 AudioMedia 的对象 audio 和指针 amPtr 以及类 VideoMedia 的对象 video。用基类指针 mPtr 指向基类对象 medium 时,可以通过 mPtr 调用定义于基类中的成员函数(表达式 mPtr－＞getName(strBuffer)和 mPtr－＞getCompany(strBuffer))。当用基类指针 mPtr 指向派生类对象 audio 时,存在隐式的类型转换,即从指针 mPtr 的角度看,对象 audio 的类型是 Media 类型,通过指针 mPtr 只能访问到定义于基类 Media 中的成员函数,如果想要调用定义于派生类中的成员函数,则必须对指针进行类型转换,只有将指针 mPtr 显式转换为派生类指针类型,才可以通过该指针访问定义于派生类中的成员函数,类型转换方式如下:

```
((AudioMedia * )mPtr)->getSinger(strBuffer);
```

用派生类指针操作基类对象时,需要进行类型的强制转换,因为基类对象不能直接当作派生类对象使用。类型转换的格式如下:

```
amPtr=(AudioMedia * )&medium;
```

用派生类指针操作基类对象实际上就是把基类对象当作派生类对象使用,这存在潜在的危险。如果通过该指针访问定义于基类中的成员,程序不会出错,因为基类对象中拥有这些成员,但如果通过该指针访问定义于派生类中的成员,由于基类对象中没有这些成员,将导致访问越界,程序会出现不可预测的错误。例如,程序中的语句:

```
amPtr->getSinger(strBuffer);
```

调用了 ampPtr 所指向对象的 getSinger 成员函数,实际则访问该对象中的数据成员 singer,由于对象 medium 中没有数据成员 singer,程序访问越界,该函数调用返回了一个不可预知的数据(字符串"?↕")。有时这种越界错误甚至可能导致程序的非正常退出。

下面总结基类指针和派生类指针在操作基类对象或派生类对象时,存在的 4 种可能的匹配方式。

(1) 直接用基类指针操作基类对象。不需要类型转换,通过基类指针可以直接访问对象中所有公有成员。

(2) 使用基类指针操作派生类对象。存在隐式的类型转换,即将派生类对象隐式转换为基类对象,但通过基类指针只能访问对象中定义于基类的公有成员。如果要访问定义于派生类中的成员,则必须将基类指针强制转换为派生类指针,直接使用基类指针访问派生类成员是一种语法错误(编译器会发现这类错误)。

(3) 使用派生类指针操作基类对象。用派生类指针操作基类对象时必须将基类对象强制转换为派生类类型。派生类指针可以访问定义于基类或派生类中的所有公有成员,但访问定义于派生类中的成员时,由于基类对象中不包含该成员,虽然编译不会出错,但会出现运行错误。

(4) 直接用派生类指针引用派生类对象。不需要类型转换,通过派生类指针可以访问对象中定义于基类和派生类中的所有公有成员。

常用的程序设计方法是通过数组或链表来统一组织和管理所有基类对象和各派生类的对象,然后用基类指针来操作它们。这种方法很方便,但也会带来麻烦。例如,如果当前对象是派生类对象,而且需要访问其定义于派生类中的成员时,必须使用类型的强制转换,将指针临时转换成派生类类型。第15章将介绍新的方法——多态性,使用该方法,用基类指针操作基类对象和派生类对象时,不需进行类型转换就可根据指针所指对象的实际类型,灵活地确定到底是访问定义于基类中的成员函数还是定义于派生类中的成员函数。

14.4 继承中的构造函数和析构函数

在继承的过程中,基类的构造函数和析构函数不被派生类继承,基类有属于自己的构造函数和析构函数,其函数名和类名相同(析构函数名比类名多了一个符号~),派生类也有自己的构造函数和析构函数。构造函数和析构函数虽然不能被继承,但派生类和基类的构造函数和析构函数之间存在一种自动调用的关系。

由于派生类继承了基类的成员,派生类的构造函数需要调用基类的构造函数对其中定义于基类的数据成员进行初始化。如果基类的构造函数不需要参数,派生类的构造函数可以隐式地调用基类的构造函数(这个过程是自动的),否则需要在派生类的构造函数中显式地调用基类的构造函数,并为它传递参数(如例14-4)。

除了构造函数和析构函数外,派生类也不能继承基类中的赋值运算符函数,派生类中的赋值运算符函数也会隐式地调用基类中的赋值运算符函数。

派生类的构造函数总是先调用基类的构造函数来初始化派生类对象中从基类继承下来的数据成员,然后才执行自己的函数体来初始化定义于派生类的数据成员。析构函数的执行顺序则正好相反,派生类的析构函数总是先执行自己的函数体,然后再调用基类的析构函数。因此,基类的构造函数总是在派生类的构造函数之前执行,而基类的析构函数则总是在派生类的析构函数之后执行。

【例14-6】 定义基类People和派生类Teacher,要求输出它们的构造函数和析构函数的执行信息,然后分别声明它们的实例化对象以检测继承关系中构造函数和析构函数的执行顺序。

```
//文件 people.h: 定义类 People
#if !defined __PEOPLE__H__
#define __PEOPLE__H__

class People
{
public:
```

```
    People(char * str);                 //构造函数
    ~People();                          //析构函数
protected:
    char * name;
};
#endif
//文件 people.cpp: 类 People 的实现
#include <string.h>
#include <iostream.h>
#include "people.h"
//构造函数的实现
People::People(char * str)
{
    name= new char[strlen(str)+ 1];
    strcpy(name,str);
    //输出构造函数执行信息
    cout<<"People construct: "<<name<<endl;
}
//析构函数的实现
People::~People()
{
    //输出析构函数执行信息
    cout<<"People destroy: "<<name<<endl;
    delete []name;
}
//文件 teacher.h: 定义类 Teacher
#include "people.h"
#if !defined __TEACHER__H__
#define __TEACHER__H__
class Teacher: public People
{
public:
    Teacher(char * str,char * sch);                   //构造函数
    ~Teacher();                                       //析构函数
protected:
    char * school;
};
#endif
//文件 teacher.cpp: 类 Teacher 的实现
#include <string.h>
#include <iostream.h>
```

```
#include "teacher.h"
//构造函数
Teacher::Teacher(char * str,char * sch): People(str)          //调用基类的构造函数
{
    school= new char[strlen(sch)+ 1];
    strcpy(school,sch);
    //输出构造函数执行信息
    cout<< "Teacher construct: "<< name<< " in "<< school<< endl;
}
//析构函数
Teacher::~ Teacher()
{
    //输出析构函数执行信息
    cout<< "Teacher destroy: "<< name<< " in "<< school<< endl;
    delete []school;
}
//文件 ex14- 6.cpp: 测试继承中基类和派生类的构造函数与析构函数的执行顺序
#include <iostream.h>
#include "teacher.h"
main()
{
    {
        People tmp("Zhang San");
    }
    People p("Li Si");
    Teacher t("Wang Wu","Wuhan University");
    return 0;
}
```

运行结果:

```
People construct: Zhang San
People destroy: Zhang San
People construct: Li Si
People construct: Wang Wu
Teacher construct: Wang Wu in Wuhan University
Teacher destroy: Wang Wu in Wuhan University
People destroy: Wang Wu
People destroy: Li Si
```

例 14-6 中的程序由 5 个部分组成。

第一部分是文件 people.h,定义了一个简单的基类 People,其中只包含一个构造函数、一个析构函数和一个受保护的数据成员 name。这里将数据成员 name 定义为受保护的而不是私有的,因为受保护的成员除了可以被类的成员函数和友元函数访问外,还可以被派生

类的成员函数访问，将 name 定义成受保护的，可以使派生类（类 Teacher）的构造函数和析构函数能够直接访问该据成员，如果将 name 定义为私有的，派生类的成员函数就只能通过基类的公有成员函数或受保护的成员函数去访问该数据了。可以看出，受保护这种可见性是专门为继承服务的一种访问控制属性。

第二部分是文件 people.cpp，是类 People 的实现部分，程序在类 People 的构造函数和析构函数中增加了输出语句以输出当前运行函数以及对象的信息。

第三部分是文件 teacher.h，定义了一个简单的派生类 Teacher，派生类中也只定义了一个构造函数、一个析构函数和一个受保护的数据成员 school。

第四个部分是文件 teacher.cpp，是类 Teacher 的实现部分，在其构造函数和析构函数中也增加了输出语句打印当前执行函数和对象的信息。

第五部分是文件 ex14-6.cpp，定义了测试基类 People 和派生类 Teacher 的主函数。程序首先在 main 函数的一个程序块中声明了类 People 的一个实例化对象 tmp，由于程序在进入其作用域后又马上退出其作用域。因此，程序在调用了类 People 的构造函数后立即又调用了其析构函数。之后程序声明了类 People 的一个对象 p，调用了类 People 的构造函数，输出了对象 p 的数据"Zhang San"。接着程序声明了类 Teacher 的对象 t，由于类 People 是类 Teacher 的基类，所以在这个过程中首先调用基类 People 的构造函数，输出对象 t 传递过去的参数"Wang Wu"，然后才执行派生类 Teacher 的构造函数体。主函数 main 结束时，对象 p 和 t 都要被撤销，由于对象 p 和 t 在分配空间时采用的是栈式分配，后声明的对象 t 要先释放。在继承体系中，派生类和基类的析构函数的调用顺序和构造函数的调用顺序相反。撤销对象 t 时，先执行派生类 Teacher 的析构函数体，然后调用基类 People 的析构函数，两个析构函数都输出了对象 t 的信息。最后撤销对象 p 时，直接调用基类 People 的析构函数。

*14.5　多重继承

本章前面讨论的继承都是单重继承，即一个派生类最多只能有一个基类，继承所构成的层次关系是树形结构。在 C++ 中，一个派生类也可以从多个基类派生而来，这种继承方式称为多重继承。多重继承允许一个派生类同时继承多个基类中的成员，支持了软件的重用性，但也可能带来大量的二义性问题。

多重继承的正确使用可以使程序具有更大的灵活性，它允许一个类型与其他多个类型之间存在“是(is_a)”的关系，即类型 A 可以“是”类型 B 同时也“是”类型 C。

例 14-7 至例 14-11 6 个部分中的程序是一个多重继承的例子。在这个例子中定义了两个基类：学生类 Student 和军人类 ArmyMan，然后定义了一个派生类 Cadet（军校学员），这个类同时继承了类 Student 和类 ArmyMan。

第一部分（例 14-7）和第二部分（例 14-8）是类 Student 的定义和实现，类 Student 包含一个受保护的数据成员 name 和一个公有成员函数 getName。在类 Student 的实现部分，构造函数在参数前面加上了称谓"Mr."，而成员函数 getName 返回受保护成员 name 的值。

第三部分（例 14-9）是类 ArmyMan 的定义，类 ArmyMan 和类 Student 相似，只包含一个受保护的数据成员 name 和一个公有成员函数 getName。

第四部分(例14-10)是基类ArmyMan的实现,和类Student的实现相似,只是构造函数在参数前面加上了称谓"Lt.",其成员函数getName返回受保护的数据成员name。

第五部分(例14-11)和第六部分(例14-12)是军人学员类Cadet的定义与实现。类Cadet通过多重继承机制继承了类ArmyMan和类Student,它没有定义数据成员,只包含两个成员函数getStudentName和getArmyManName。但由于它继承了类ArmyMan和类Student的所有成员,所以类Cadet拥有两个数据成员name和两个成员函数getName,分别来自于两个不同的基类。

【例14-7】 定义基类Student。

```
//文件Student.h:定义学生类Student
#if !defined __STUDENT__H__
#define __STUDENT__H__

class Student {
public:
    Student(char * );
    ~Student();
    char * getName(char * );

protected:
    char * name;
};

#endif
```

【例14-8】 基类Student的实现。

```
//文件student.cpp:类Student的实现
# include "student.h"
# include <string.h>

Student::Student(char * studName)
{
    char * title="Mr. ";
    name=new char[strlen(title)+strlen(studName)+1];
    strcpy(name,title);
    strcat(name,studName);
}

Student::~Student()
{
    delete [] name;
}

char * Student::getName(char * str)
{
    strcpy(str,name);
    return str;
}
```

【例 14-9】 定义基类 ArmyMan。

```
//文件 ArmyMan.h:定义军人类 ArmyMan
#if !defined __ARMYMAN__H__
#define __ARMYMAN__H__

class ArmyMan {
public:
    ArmyMan(char* );
    ~ArmyMan();

    char * getName(char* );

protected:
    char * name;
};

#endif
```

【例 14-10】 基类 ArmyMan 的实现。

```
//文件 ArmyMan.cpp:类 ArmyMan 的实现
# include "armyman.h"
# include <string.h>

ArmyMan::ArmyMan(char * armyManName)
{
    char * title=" Lt.";
    name=new char[strlen(title)+strlen(armyManName)+1];
    strcpy(name,title);
    strcat(name,armyManName);
}

ArmyMan::~ArmyMan()
{
    delete [] name;
}

char * ArmyMan::getName(char * str)
{
    strcpy(str,name);
    return str;
}
```

多重继承的语法非常简单，在类的定义中类名的后面通过冒号引出多个基类，每个基类可以分别设置继承属性(本例中都是 public)，多个基类之间用逗号隔开。

【例 14-11】 定义派生类 Cadet。

```
//文件 cadet.h:定义军人学员类 Cadet
#if !defined __CADET__H__
#define __CADET__H__
```

```
#include "armyman.h"
#include "student.h"
//军人学员类Cadet继承了类ArmyMan和类Student
class Cadet: public ArmyMan,public Student{
public:
    Cadet(char* );
    char * getStudentName(char* );
    char * getArmyManName(char* );
};
#endif
```

在下面类Cadet的实现部分,构造函数显式的调用了两个基类的构造函数,并把同一个参数n分别传递给了两个基类的构造函数。类Cadet的成员函数getStudentName希望返回定义于基类Student中的数据成员name,由于类Cadet同时拥有两个数据成员name(分别来自于两个基类),如果直接用成员名去访问就存在二义性,要解决这个问题,只需要在成员名前面加上类名和二目作用域运算符即可。例如,Student::name表示访问定义于基类Student中的数据成员name,ArmyMan::getName()则表示调用定义于基类ArmyMan中的成员函数getName。

【例14-12】 派生类Cadet的实现。

```
//文件Cadet.cpp: 类Cadet的实现
# include "cadet.h"
# include <string.h>
//类Cadet的构造函数显式调用了两个基类的构造函数
Cadet::Cadet(char * n)
:Student(n),ArmyMan(n)                    //显式调用基类构造函数
{
}
char * Cadet::getStudentName(char * str)
{
    //访问基类Student定义的成员
    strcpy(str,Student::name);
    return str;
}
char * Cadet::getArmyManName(char * str)
{
    //访问基类ArmyMan定义的成员
    strcpy(str,ArmyMan::name);
    return str;
}
```

定义完上述3个类之后,例14-13对它们进行了测试。例14-13中的程序需要包含头文件cadet.h,并且将上述3个类的定义和实现文件加入到工程中。例14-13中的程序首先声

明了类 Cadet 的实例化对象 zhang,并将字符串"Zhang San"传递给类 Cadet 的构造函数,类 Cadet 的构造函数又分别将该字符串传递给了类 Student 和类 ArmyMan 的构造函数,对象 zhang 中的两个数据成员 name(分别来自于两个基类)分别被初始化为"Mr. Zhang San"和"Lt. Zhang San"。

接下来,程序通过对象 zhang 调用了定义于基类 ArmyMan 中的成员函数 getName,由于类 Cadet 中存在两个 getName 函数,直接使用成员函数名去调用该函数存在二义性,因此也需要加上类名和二目作用域运算符,格式为 zhang. ArmyMan::getName()。调用定义于 ArmyMan 中的成员函数 getName 返回的是定义于 ArmyMan 中的数据成员 name 中的字符串。然后,程序又通过调用定义于 Cadet 中的成员函数 getArmyManName 来输出定义于 ArmyMan 中的数据成员 name。程序也通过类似的方法访问了定义于基类 Student 中的成员函数和数据成员。

【例 14-13】 编写程序使用上面例 14-11 定义的具有多重继承关系的 3 个类,创建派生类 Cadet 的实例化对象并调用其成员函数,注意其中调用存在歧义的函数的语法。

```
//文件 ex14_13.cpp: 测试多重继承的类层次
#include <iostream.h>
#include "cadet.h"
#define MAX_STR_LEN 256
int main()
{
    Cadet zhang("Zhang San");
    char name[NAME_MAX_LEN];
    //调用基类 ArmyMan 中定义的成员函数 getName
    cout<<"Call getName() in ArmyMan: "<<zhang.ArmyMan::getName(name)<<endl;
    cout<<"Call getArmyManName(): "<<zhang.getArmyManName(name)<<endl;
    //调用基类 Student 中定义的成员函数 getName
    cout<<"Call getName() in Student: "<<zhang.Student::getName(name)<<endl;
    cout<<"Call getStudentName(): "<<zhang.getStudentName(name)<<endl;
    return 0;
}
```

运行结果:

```
Call getName() in ArmyMan: Lt. Zhang San
Call getArmyManName(): Lt. Zhang San
Call getName() in Student: Mr. Zhang San
Call getStudentName(): Mr. Zhang San
```

14.6　软件渐增式开发

面向对象的软件开发方法一般是从分析问题模型开始,然后识别出一个个对象,并对它们进行细化、抽象,最后用程序语言去描述它们的过程。这种软件开发方法从本质上来讲是

迭代的和渐增的,其过程就是一次次的迭代反复的过程。随着迭代的进行,系统的功能不断完善。和传统的瀑布式软件开发模型相比,面向对象软件开发在分析、设计和编码等各个阶段之间的界限变得模糊起来。因为面向对象软件开发关心的是对象以及对象之间的关系。与结构化程序设计的思想相比,面向对象软件开发的重心从编码向分析偏移,从以功能为中心向以数据为中心偏移。而且,面向对象软件开发中的继承和复合使得软件的重用变得更加自然。

14.6.1 复合与继承

对象之间通过继承实现了“是(is_a)”的关系,通过复合可实现把对象作为另外一个类的成员的“拥有(has_a)”的关系。“拥有(has_a)”关系通过包含(复合)现有类的对象构造新类。例如,圆柱类 Column 和圆类 Circle,如果说圆柱就是一个圆显然不合适,但如果说圆柱中包含了圆就合适了。“是(is_a)”的关系则通过继承现有类(基类)中的成员来构造新类。例如,前面定义的类 Mammal 和 Lion、类 People 和 Teacher 等,一个 Lion 可以是一个 Viviparity, Viviparity 适用的一些属性和操作同样适合于 Lion,一个 Teacher 也可以作为一个 People,对 People 具有的属性和操作也同样适合于 Teacher。因此,在已建立类 Viviparity 和 People 的前提下,可通过继承的方式建立类 Lion 和 Teacher。

无论复合还是继承,都能使得新的类包含或包容已有类的属性和操作。在程序设计时应该如何选择呢？一般情况下,如果在新的类中有成员是已有类的对象,使用复合方式比较合适;如果新的类可以视为已有类的子类或特例,新类的对象也可以作为已有类的对象使用,那么使用继承更合适。

除了复合和继承外,对象之间还可以建立“使用”和“知道”的关系,“使用”关系是指一个对象可以获取另一个对象的句柄(对象名、对象引用或对象指针),然后调用该对象的成员函数为自己服务。如读者(Reader)要借书时,不能自己到图书馆去操作,而只能请求图书管理员为自己服务,完成借书的操作(调用图书管理员的相应的成员函数),这里读者就“使用”了图书管理员。

一个对象也可以“知道”另一个对象,这种关系在知识表示中经常用到。一个对象可以拥有指向另一个对象的指针或对象的引用,从而“知道”另一个对象的存在。

14.6.2 示例

下面分析一个类的复合和继承的综合示例。在该例中,首先定义并实现了点类 Point,定义了一个线段类 Line,类 Line 以复合的方式使用了类 Point,Line 的表示端点的两个成员 startPoint 和 endPoint 都是 Point 的实例;然后定义了多边形类 Polygon,Polygon 使用了 Line,Polygon 表示边的成员 borders 是一个 Line 对象数组,这也是一种复合方式;最后从类 Polygon 派生出子类 Triangle,Triangle 是从 Polygon 按照公有方式派生的。图 14-2 给出了各类之间的关系。下面为每一个类分别编写测试程序。

点、线段、多边形之间是组成或“拥有(has_a)”的关系:一个线段包含两个端点,一个多边形由多条边(线段)组成,所以采取了类的复合的方式。而三角形是多边形的特例,三角形本身也是多边形,这是一种典型的“是(is_a)”关系,所以采取了继承方式。

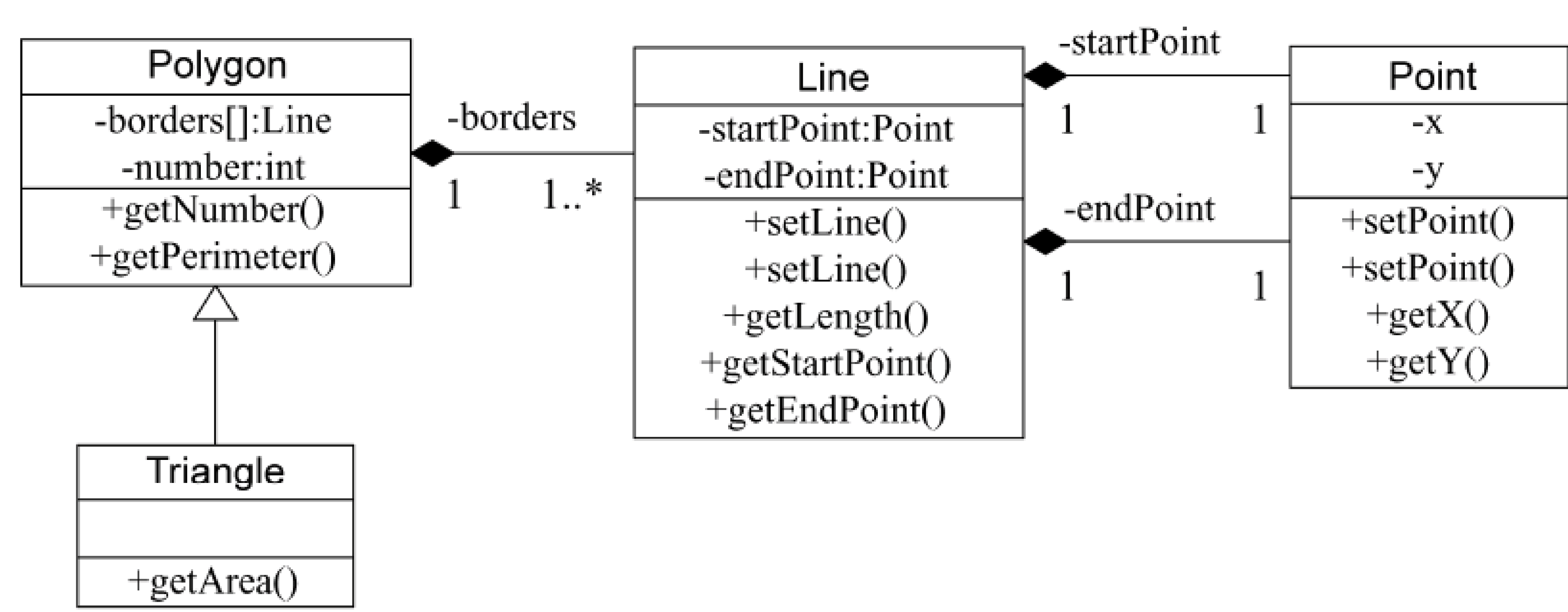


图 14-2 各类之间的关系

下面的例 14-14 和例 14-15 给出了类 Point 的定义和实现，其中第一部分（例 14-14）是类 Point 的定义。为了实现数据的封装和信息隐藏，类 Point 表示点的坐标的数据成员是私有的，但是提供了对它们定值和取值的公有函数。为了使用方便，类的构造函数和定值函数都进行了重载，特别是提供了复制构造函数。为了输出对象的信息，还为类 Point 重载了流插入运算符（<<）用于对象信息的输出。

【例 14-14】 类 Point 的定义。

```
//文件 Point.h：类 Point 的定义
#if !defined __POINT__H__
#define __POINT__H__

#include <iostream.h>
class Point
{
    friend ostream &operator<< (ostream &,const Point &);
public:
    //重载的构造函数
    Point(double=0,double=0);
    Point(const Point &p);             //复制构造函数

    //析构函数
    ~ Point();

    //重载的定值函数
    void setPoint(double a,double b);
    void setPoint(Point &p);

    //取值函数
    double getX()const
    {
        return x;
    } double getY()const
    {
        return y;
    }
```

```
private:
    double x,y;
};
#endif
```


第二部分(例14-15)是类Point的实现程序。类Point的构造函数和setPoint成员函数的实现都非常简单,直接对数据成员x和y进行赋值。流插入运算符重载函数按一定的格式输出了点的坐标。

【例14-15】 类Point的实现。

```
//文件 point.cpp:类 Point的实现
#include "point.h"
Point::Point(double a,double b)
{
    x=a;
    y=b;
}
Point::Point(const Point &p)
{
    x=p.getX();
    y=p.getY();
}
Point::~Point(){}
void Point::setPoint(double a,double b)
{
    x=a;
    y=b;
}
void Point::setPoint(Point &p)
{
    x=p.getX();
    y=p.getY();
}
ostream &operator<< (ostream &os,const Point &point)
{
    os<<"("<<point.x<<","<<point.y<<")";
    return os;
}
```

下面示例中的程序是类Point的测试程序。在main函数中,首先声明了对象P,然后调用类Point的友元函数operator <<输出对象P的位置信息,受保护的成员可以被成员函数和友元函数直接访问。之后,程序调用setPoint函数修改了对象P的值,然后调用get函数输出对象P新的位置。在主函数中,如果要访问类的受保护成员,必须通过调用其公

有成员函数进行。

【例 14-16】 编写程序使用例 14-14 中定义的类 Point,创建类 Point 的对象并输出其信息。

```
//文件 ex14_16.cpp: 使用类 Point
# include "Point.h"

int main()
{
    //声明类 Point 的对象 P
    Point p(3.0,3.5);

    cout<< "p: "<< p<< endl;

    p.setPoint(6.5,5.5);

    cout<< "After setPoint:\n";
    cout<< "p: "<< "("<< p.getX()<< ","<< p.getY()<< ")"<< endl;

    return 0;
}
```

运行结果:

```
p: (3,3.5)
After setPoint:
p: (6.5,5.5)
```

第二个例子是类 Line。这个例子使用了例 14-14 中的类 Point。例 14-17 和例 14-18 分别是类 Line 的定义和实现。类 Line 以复合方式使用类 Point,Line 有表示端点的两个私有成员 startPoint 和 endPoint,它们都是 Point 对象。同样,为了方便类的使用者,类 Line 也提供了复制构造函数,重载了定值函数,并重载了流插入运算符(<<)用于类 Line 对象的输出。此外,针对线段的特殊需求,还提供了计算线段长度的函数。值得注意的是类 Line 的构造函数调用成员对象的构造函数的方式。

重载运算符<<时,程序首先利用 Point 定义的流插入运算符重载函数,直接输出了 startPoint 和 endPoint,然后输出线段的长度。

例 14-19 是在例 14-16 的基础上扩充实现的,演示了对类 Point 和 Line 的使用。注意主函数使用了两种方法构造 Line 类型的对象。

【例 14-17】 类 Line 的定义

```
//文件 line.h: 类 Line 的定义
# if !defined __LINE__H__
# define __LINE__H__

# include <iostream.h>
# include "point.h"

class Line {
    friend ostream & operator<< (ostream &,const Line &);
public:
```

```
        //重载的构造函数
        Line(double startX=0,double startY=0,double endX=0,double endY=0);
        Line(Point start,Point end);
        Line(Line &line);                          //复制构造函数
        //析构函数
        ~Line();
        //重载的定值函数
        void setLine(double startX=0,double startY=0,double endX=0,double endY=0);
        void setLine(Point,Point);
        double getLength() const;                  //计算线段的长度
        //取值函数
        Point getStartPoint() {return startPoint;}
        Point getEndPoint() {return endPoint;}
    private:
        Point startPoint,endPoint;
    };
    #endif
```

【例 14-18】 类 Line 的实现程序。

```
//文件 line.cpp: 类 Line 的实现
#include <math.h>
#include "line.h"
Line::Line(double startX,double startY,double endX,double endY)
: startPoint(startX,startY),endPoint(endX,endY){}
Line::Line(Line &line)
: startPoint(line.getStartPoint()),endPoint(line.getEndPoint())
//利用 Point 的复制构造函数
{}
Line::Line(Point start,Point end)
: startPoint(start),endPoint(end)           //利用 Point 的复制构造函数
{}
Line::~Line(){}
void Line::setLine(double startX,double startY,double endX,double endY)
{
    startPoint.setPoint(startX,startY);
    endPoint.setPoint(endX,endY);
}
void Line::setLine(Point start,Point end)
{
    startPoint.setPoint(start);
    endPoint.setPoint(end);
```

```
}
//计算线段的长度
double Line::getLength()const
{
    double x1=startPoint.getX();
    double y1=startPoint.getY();
    double x2=endPoint.getX();
    double y2=endPoint.getY();

    return sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
}

ostream &operator<<(ostream &os,const Line &line)
{
    os<<line.startPoint<<"->"<<line.endPoint<<". ";
    os<<"Its length is: "<<line.getLength();

    return os;
}
```

【例 14-19】 编写程序使用例 14-17 和例 14-18 中定义的类 Point 和 Line,分别创建它们的实例化对象并输出对象信息。

```
//文件 ex14_19.cpp
#include "point.h"
#include "line.h"

int main()
{
    //声明并输出两个点
    Point p1(3.0,3.5);
    Point p2(6.5,5.5);

    cout<<"p1: "<<p1<<endl;
    cout<<"p2: "<<p2<<endl;

    cout<<endl;

    //声明并输出两条线段
    Line l1(p1,p2);
    Line l2(1.0,2.5,4.5,5.0);

    cout<<"l1: "<<l1<<endl;
    cout<<"l2: "<<l2<<endl;

    return 0;
}
```

运行结果：

```
p1: (3,3.5)
p2: (6.5,5.5)
```

```
l1: (3,3.5) -> (6.5,5.5). Its length is: 4.03113
l2: (1,2.5) -> (4.5,5). Its length is: 4.30116
```

下面仍然使用复合的方法,构造多边形类Polygon。类Polygon使用了Line数组borders来表示多边形的边,其构造函数利用参数提供的Point数组来设置每条边的坐标。根据多边形的特殊性,提供了计算多边形周长的方法。同样地,通过重载<<运算符,提供了Polygon对象的输出方法。

【例14-20】 类Polygon的定义。

```
//文件 polygon.h: 类 Polygon 的定义
#if !defined __POLYGON__H__
#define __POLYGON__H__

#include <iostream.h>
#include "line.h"

#define MAX_BORDER_NUM 6
class Polygon
{
    friend ostream &operator<< (ostream &,const Polygon &);
public:
    Polygon(Point points[],int num);              //构造函数
    ~Polygon();                                   //析构函数
    int getNumber()const

    {
        return number;
    }
    double getPerimeter()const;                   //计算多边形的周长

protected:
    Line borders[MAX_BORDER_NUM];
    int number;
};

#endif
```

【例14-21】 类Polygon的实现。

```
//文件 polygon.cpp: 类 Polygon 的实现
#include "polygon.h"

Polygon::Polygon(Point points[],int num)
{
    number=num;
    for (int i=0; i<num; i++)
        borders[i].setLine(points[i],points[(i+1) %num]);
}

Polygon::~Polygon(){}

//计算多边形的周长
```

```
double Polygon::getPerimeter() const
{
    double perimeter=0.0;
    for (int i=0; i<number; i++)
        perimeter+=borders[i].getLength();
    return perimeter;
}

ostream &operator<< (ostream &os,const Polygon &polygon)
{
    os<<"The polygon has "<<polygon.number<<" borders:";
    for (int i=0; i<polygon.number; i++)
        os<<endl<<"\tborder["<<i<<"]: "<<polygon.borders[i];
    os<<endl<<"\tIts perimeter is: "<<polygon.getPerimeter()<<endl;
    return os;
}
```

在例14-19中增加对类Polygon的对象的操作，就得到了下面的示例。

【例14-22】 编写程序使用本节上面各例子中定义的3个类Point、Line和Polygon，分别创建它们的实例化对象并输出对象信息。

```
//文件 ex14_22.cpp
# include "polygon.h"

int main()
{
    //声明并输出两个点
    Point p1(3.0,3.5);
    Point p2(6.5,5.5);

    cout<<"p1: "<<p1<<endl;
    cout<<"p2: "<<p2<<endl;

    cout<<endl;

    //声明并输出两条线段
    Line l1(p1,p2);
    Line l2(1.0,2.5,4.5,5.0);

    cout<<"l1: "<<l1<<endl;
    cout<<"l2: "<<l2<<endl;

    cout<<endl;

    //声明并输出一个多边形(五边形)
    Point points5[5];
    points5[0].setPoint(0,0);
    points5[1].setPoint(0,1);
    points5[2].setPoint(1,2);
    points5[3].setPoint(2.5,2);
    points5[4].setPoint(2,0.5);
```

```
    Polygon ply(points5,5);
    cout<<"ply: "<<ply<<endl;
    return 0;
}
```

运行结果:

```
p1: (3,3.5)
p2: (6.5,5.5)
l1: (3,3.5)->(6.5,5.5). Its length is 4.03113
l2: (1,2.5)->(4.5,5). Its length is 4.30116
ply: The polygon has 5 border(s):
        border[0]: (0,0)->(0,1). Its length is 1
        border[1]: (0,1)->(1,2). Its length is 1.41421
        border[2]: (1,2)->(2.5,2). Its length is 1.5
        border[3]: (2.5,2)->(2,0.5). Its length is 1.58114
        border[4]: (2,0.5)->(0,0). Its length is 2.06155
        Its perimeter is: 7.55691
```

最后,使用继承的方式,利用多边形类 Polygon 建立新的三角形类 Triangle。类 Triangle 以公有方式继承了类 Polygon。这样 Triangle 继承了 Polygon 的属性和方法。例如,计算周长的 getPerimeter 函数。Triangle 没有增加新的数据成员,但是增加了新的函数 getArea,用来计算三角形的面积。Triangle 有自己的流插入运算符(<<)重载函数,输出 Triangle 对象信息时,首先输出提示信息,再将 Triangle 对象作为 Polygon 对象输出每条边和周长的信息,最后输出面积。

例 14-25 中的程序使用了本节所定义的所有类,并对它们的实例化对象进行了测试。

【例 14-23】 类 Triangle 的定义。

```
//文件 triangle.h: 类 Triangle 的定义
#if !defined __TRIANGLE__H__
#define __TRIANGLE__H__
#include <iostream.h>
#include "polygon.h"
class Triangle: public Polygon
{
    friend ostream &operator<< (ostream &,const Triangle &);
public:
    Triangle(Point points[]);            //构造函数
    ~Triangle();                         //析构函数
    double getArea()const;               //计算三角形的面积
};
#endif
```

【例 14-24】 类 Triangle 的实现。

```
//文件 triangle.cpp: 类 Triangle 的实现
#include <math.h>
#include "triangle.h"

Triangle::Triangle(Point points[]): Polygon(points,3){}

Triangle::~Triangle(){}

double Triangle::getArea()const
{
    double a=borders[0].getLength();
    double b=borders[1].getLength();
    double c=borders[2].getLength();
    double s=(a+b+c)/2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}

ostream &operator<<(ostream &os,const Triangle &tri)
{
    os<<"Triangle is Polygon. ";
    os<<(const Polygon &)tri;                    //将 Triangle 作为 Polygon 输出
    os<<"\tIt's area is: "<<tri.getArea()<<endl;

    return os;
}
```

【例 14-25】 编写程序使用本节上述各例子中定义的 4 个类 Point、Line、Polygon 和 Triangle，分别创建它们的实例化对象并输出对象信息。

```
//文件 ex14_25.cpp
#include "Triangle.h"
main()
{
    //声明并输出两个点
    Point p1(3.0,3.5);
    Point p2(6.5,5.5);

    cout<<"p1: "<<p1<<endl;
    cout<<"p2: "<<p2<<endl;

    cout<<endl;

    //声明并输出两条线段
    Line l1(p1,p2);
    Line l2(1.0,2.5,4.5,5.0);

    cout<<"l1: "<<l1<<endl;
    cout<<"l2: "<<l2<<endl;

    cout<<endl;

    //声明并输出一个多边形(五边形)
```

```
        Point points5[5];
        points5[0].setPoint(0,0);
        points5[1].setPoint(0,1);
        points5[2].setPoint(1,2);
        points5[3].setPoint(2.5,2);
        points5[4].setPoint(2,0.5);
        Polygon ply(points5,5);

        cout<<"ply: "<<ply<<endl;

        cout<<endl;
        //声明并输出一个三角形
        Point points3[3];
        points3[0].setPoint(1,1);
        points3[1].setPoint(3,3);
        points3[2].setPoint(1,3);
        Triangle tri(points3);

        cout<<"tri: "<<tri<<endl;

        return 0;
}
```

运行结果:

```
p1: (3,3.5)
p2: (6.5,5.5)

l1: (3,3.5) -> (6.5,5.5). Its length is: 4.03113
l2: (1,2.5) -> (4.5,5). Its length is: 4.30116

ply: The polygon has 5 borders:
        border[0]: (0,0) -> (0,1). Its length is: 1
        border[1]: (0,1) -> (1,2). Its length is: 1.41421
        border[2]: (1,2) -> (2.5,2). Its length is: 1.5
        border[3]: (2.5,2) -> (2,0.5). Its length is: 1.58114
        border[4]: (2,0.5) -> (0,0). Its length is: 2.06155
        Its perimeter is: 7.55691

tri: Triangle is Polygon. The polygon has 3 borders:
        border[0]: (1,1) -> (3,3). Its length is: 2.82843
        border[1]: (3,3) -> (1,3). Its length is: 2
        border[2]: (1,3) -> (1,1). Its length is: 2
        Its perimeter is: 6.82843
        It's area is: 2
```

这个例子综合应用了类的复合与继承、派生类对象和基类成员的转换、运算符重载等知识。通过类的复合与继承,可以利用已有的类来构造新的类,就如同搭积木一样,这对于提高软件的重用水平、以渐进的方式开发大型的软件提供了很好的支持。下一章将要讨论是如何在继承的层次结构中使用多态性编写更好的程序。数据抽象、继承和多态性是面向对象程序设计的关键所在。

习　题　14

14.1　填空：

(1) 派生类可以定义其________中不具备的数据和操作。

(2) 如果在派生类中要访问被重定义了的基类同名函数，那么需使用________才可调用。

(3) 要把基类指针作为派生类指针处理，必须使用________。指向基类的对象的指针变量也可以指向________的对象。

(4) 创建一个派生类的对象时，需要先调用________的构造函数，后调用________的构造函数。

(5) C++ 的________允许一个派生类继承多个基类。

(6) 类之间的"有"关系代表________，"是"关系代表________。

14.2　当从一个类派生出新类时，可以对派生类进行哪些变化？

14.3　编写一个教室类 Classroom，然后描述其派生类多媒体教室 MMClassroom，多媒体教室是增加了多媒体教学设备的教室。

14.4　编写一个笔类，描述所有笔的共同属性，然后编写笔类的各派生类：钢笔、铅笔、签字笔、毛笔，在各派生类中尽量描述清楚各自的属性。

14.5　下面是一个形状类 Shape，编写类 Shape 的派生类：圆类 Circle、三角形类 Triangle 和矩形类 Rectangle，并重定义基类的成员函数使之返回正确的结果(show 函数要输出对象的基本信息)，然后编写程序测试它们。

```
class Shape {
public:
    //形状的面积
    double area() {
        return 0;
    }
    //形状的周长
    double girth() {
        return 0;
    }
    //输出对象的信息
    void show() {
        cout<< "Shape Object:"<< endl;
    }
private:
};
```

14.6　在 14.6.2 节程序的基础上，定义 Polygon 新的子类矩形类 Rectangle，要求在 Rectangle 中实现其面积计算功能，并提供相应的输出函数。编写一个测试程序，演示类的功能。

第 15 章 多态性

【学习内容】

本章介绍多态性。主要内容包括：

◆ 静态绑定和动态绑定的概念。
◆ 多态性的意义和作用。
◆ 虚函数的定义方法。
◆ 抽象基类的作用和定义方法。
◆ 虚析构函数的作用和使用方法。
◆ 如何在程序中实现多态性。

【学习目标】

◆ 理解静态绑定和动态绑定的概念。
◆ 了解多态性的意义和作用。
◆ 掌握定义虚函数和抽象基类的方法。
◆ 掌握实现多态性的方法以及使用多态性的时机。
◆ 了解虚析构函数的作用。
◆ 掌握如何使用多态性进行软件的渐增式开发。

多态性是面向对象程序设计的重要特性之一，它使得设计和实现易于扩展的系统成为可能。利用多态性的机制，即使在一个类还没有定义而类的对象还不存在时，就可以先编写使用和处理这些对象的程序。这时，唯一需要做的就是要为这些类定义基类，并提供操作基类对象的通用框架。这样的程序可以留待自己将来或其他人根据需要进行扩展（定义新的派生类以及派生类的成员函数）。本章将介绍多态性的概念及使用方法。

15.1 多态性的概念

面向对象程序一般需要创建很多对象并协调它们共同完成程序所需要的功能。有时需要在程序中操作多种不同类型的对象，完成对不同对象的不同处理。为此，常用手段是使用分支语句。通过分支语句判断对象的不同类型以决定采取什么操作。但这种方法有很多缺陷，分支语句必须对所有类型进行测试，必须保证类型测试和相应操作的一致性。另外，如果要加入新的类型就必须修改程序。

多态性可以很好地解决上述问题。在面向对象方法中,多态性是指同一操作作用于不同类的实例时,将导致不同的执行结果,即不同类的对象接收到相同的消息时,将有不同的处理方式。在C++中,通过多态性,因继承而相关的不同的类,它们的对象可以对同一个函数调用作出不同的响应。"同一个函数调用"是指同一条函数调用语句,"不同的响应"是指执行了不同的函数。使用多态性可以在不需要判断对象类型以及不修改程序的情况下正确地操作不同类型的对象(使用指针或对象引用)。

15.1.1 静态绑定和动态绑定

多态性是怎样解决上面的问题的呢?

对于前面所接触到的程序,任何一条语句都应该明确地知道它的作用:声明变量、输入输出、运算、函数调用等。对于函数调用也应该明确地知道这条语句调用的是哪一个函数,程序执行到该语句时会跳转到哪一行代码上执行,编译器可以为每一条函数调用语句确定它将执行的函数体,确定要跳转到的代码的绝对地址。这种在编译时就能确定函数调用语句和实际执行的函数关系的机制,就是所谓的"静态绑定"。

静态绑定并不能很好地解决所有问题,特别是在使用面向对象的思想进行编程的时候。例如,在一个企业里,司机、程序员、业务员、打字员都可以是一个个对象,企业领导(也是一个对象)希望他的员工都去工作,他可能只需说一句话:"大家工作吧!",这在程序里可以表示为一条消息或一个函数调用。如果采用静态绑定的方法,对于一个函数调用,只能明确地执行某一段特定的代码,这就意味着,不管是司机还是程序员、业务员、打字员,他们接收到这条相同的命令后都会去做同一件事情,这显然不是企业领导所需要的,他所需要的是不同的工作人员在接收到同一个指示"大家工作吧!"之后,能够根据自己岗位的职责开展不同的工作。要做到这一点,或者可以采用另外一种方法:领导对司机说:"你去开车吧。",对程序员说:"你去编程吧。"……但这显然太繁琐、效率太低。另外,采取这种方式,如果企业增加了新的工作人员,领导的指示也必将随之增加,企业的管理也将越发复杂和低效。

理想的解决方法是:对同一条命令,不同的人会自动去做不同的事。也就是说,对同一个函数调用,不同的对象会执行不同的函数,这就需要"动态绑定"。为了实现动态绑定,给对象发消息(调用对象的成员函数)时,编译器暂不确定将被调用执行的代码,它只要保证被调用函数存在,并对函数进行参数列表和返回值类型的检查。当程序真正运行时,再由接收消息的对象根据自己的类型来确定应该具体调用哪个函数。这种在运行时才能确定调用的函数的机制,就称为"动态绑定"。

15.1.2 多态性的意义

从前面的例子可以看出,如果通过基类的指针或引用来调用一个成员函数,不管接收消息的对象是基类对象还是派生类对象,被调用的函数都是定义于基类中的成员函数。如果使用派生类的指针或引用来调用一个成员函数时,不管接收消息的对象是基类对象还是派生类对象,被调用的函数都是定义于派生类中的成员函数,这不是多态性。

C++的多态性是通过虚函数来实现的。只有通过基类的指针或引用来调用虚函数时,才会对被调用的函数进行动态绑定,即根据接收消息的对象的类型(而不是指针或引用的类型)来确定要调用的函数。多态性可为编程带来很大的灵活性。

多态性有助于更好地对程序进行抽象,使控制模块能专注于处理一般性问题,而把具体的操作交给执行部分去做。多态性为软件实现"同一对外接口,不同内部实现"的目标提供了支持。例如,一个学校管理软件需要显示很多不同类型的学校的信息,甚至可以包括系统运行过程中不断增加的新类型的学校,这些学校可以是小学、中学、大学等(它们都是从基类School派生而来的类)。为了能显示所有学校的信息,控制模块只需要用链表或数组管理所有这些类的对象(各个不同的学校),并用基类指针逐个指向每一个对象,通过指针向所指的对象发送一条show消息即可。函数show需要在基类School中被声明为虚函数,并在每一个派生类中被重新定义和实现。这里,控制模块不需再关心显示信息的细节,而只需要简单地发送一条消息即可,接收消息的对象会执行各自不同的成员函数(自己重定义的show函数)来响应该消息,以输出各自不同的信息。

多态性有助于提高程序的可扩展性。采用多态性的方法编程,可以把控制模块与被操作的对象分开。因此,不必重新对程序进行编译就可以在程序中添加已定义类的新对象,并能自动将新对象纳入控制;甚至不需要修改基本控制程序就可以把能响应现有消息的新类(作为已有类的派生类)以及新类的对象添加到系统中。

多态性特别适合于实现一些分层的软件系统。例如,Windows操作系统需要以多种不同的窗口形式显示不同的内容(包括文本、图像等),但不论显示的内容是什么,窗口显示的操作在某种程度上都是统一的。如果利用多态性进行设计和实现,只需要给所有的窗口类建立一个公共基类,并在基类中定义一个用于在窗口中显示信息的虚函数,然后在每一个特定的窗口类中分别实现该函数,就可以实现所有窗口统一管理和操作。如果程序想要任何窗口对象显示其内容,只需要给它发消息调用该虚函数即可,每个窗口对象接收到消息后会执行它自己的显示函数。

多态性经常需要使用基类指针操作不同的对象。通常的操作方法是用链表或数组组织所有对象,然后用基类指针去遍历它们。

15.2 虚函数

在C++中,通过基类指针或引用调用一般的成员函数时采取的都是静态绑定。下面的例子对例14-4进行了修改,只保留了一个基类和一个派生类,然后用基类指针去操作基类对象和派生类对象,并使用基类指针调用成员函数,通过输出信息显示调用过程,从运行结果看,程序没有体现多态性。

【例15-1】 编写具有继承关系的两个类,并在派生类中重定义基类的成员函数,然后用基类指针操作基类和派生类的对象,并调用该成员函数,检查该调用过程是否显示多态性。

```
//文件 media.h: 定义类 Media
#if !defined __MEDIA__H__
#define __MEDIA__H__
class Media
{
public:
```

```
    //构造函数
    Media(char * n,char * c);
    //析构函数
    ~Media();
    void showInfo();
private:
    char * name;                        //媒体名
    char * company;                     //制作公司
};
#endif
//文件 media.cpp:类 Media 的实现
#include <string.h>
#include <iostream.h>
#include "media.h"
//构造函数的实现
Media::Media(char * n,char * c)
{
    name=new char[strlen(n)+1];
    strcpy(name,n);
    company=new char[strlen(c)+1];
    strcpy(company,c);
}
//析构函数的实现
Media::~Media()
{
    if (name!=NULL)
        delete []name;
    if (company!=NULL)
        delete []company;
}
void Media::showInfo()
{
    cout<<"Name: "<<name<<endl;
    cout<<"Company: "<<company<<endl;
}
//文件 audioMedia.h:定义类 AudioMedia
#if !defined __AUDIOMEDIA__H__
#define __AUDIOMEDIA__H__
#include "media.h"
class AudioMedia: public Media
{
public:
    AudioMedia(char * n,char * c,char * s);
```

```
        ~AudioMedia();
        void showInfo();
private:
        char * singer;                                //歌手
};
#endif
//文件 audioMedia.cpp: 类 AudioMedia 的实现
#include <string.h>
#include <iostream.h>
#include "audioMedia.h"
AudioMedia::AudioMedia(char * n,char * c,char * s)
: Media(n,c)                                          //调用基类的构造函数
{
        singer=new char[strlen(s)+1];
        strcpy(singer,s);
}
AudioMedia::~AudioMedia()
{
        if (singer!=NULL)
            delete []singer;
}
void AudioMedia::showInfo()
{
        Media::showInfo();                            //调用基类被重定义的函数
        cout<<"Singer: "<<singer<<endl;
}
//文件 ex15_1.cpp: 测试使用基类指针操作继承关系中的各个对象
#include <iostream.h>
#include "media.h"
#include "audioMedia.h"
int main()
{
        Media * mPtr;
        Media medium("Slumdog Millionaire","Celador Films");
        AudioMedia audio("The Colour of My Love","Columbia","Celine Dion");
        mPtr=&medium;                                 //基类指针指向基类对象
        mPtr->showInfo();                             //用基类指针调用成员函数,静态绑定
        cout<<endl;
        mPtr=&audio;                                  //基类指针指向派生类对象
        mPtr->showInfo();                             //用基类指针调用成员函数,静态绑定
        return 0;
}
```

运行结果：

```
Name: Slumdog Millionaire
Company: Celador Films

Name: The Colour of My Love
Company: Columbia
```

例15-1中的程序包含5个文件。第一个文件media.h是基类Media的定义，描述了媒体的基本信息，并提供了输出这些基本信息的成员函数showInfo。第二个文件media.cpp是类Media的实现部分。第三个文件audioMedia.h是派生类AudioMedia的定义，类AudioMedia公有继承于类Media，并重定义了函数showInfo。第四个文件audioMedia.cpp是类AudioMedia的实现部分。第五个文件ex15_1.cpp是测试程序，在main函数中声明了一个基类的指针mPtr以及基类和派生类的对象medium和audio，然后分别用mPtr指向medium和audio，并调用成员函数showInfo。从程序的输出结果可以看到，程序始终调用的是基类的showInfo函数，并没有根据对象的不同类型分别选择执行基类或派生类的showInfo函数。所以，这种调用方式没有引起动态绑定。

在C++中，只有通过基类指针或引用调用虚函数时才能引发动态绑定。C++要求在声明虚函数时使用virtual关键字。定义一个虚成员函数，只需简单地在这个函数的函数原型前面加上关键字virtual即可。如果一个函数在基类中被声明为虚函数，那么在所有派生类中它都是虚函数，即使在派生类中该成员函数的函数原型前面没有关键字virtual。

例15-2在例15-1的基础上把成员函数show定义成了虚函数。比较例15-2中和例15-1中的类Media的定义，例15-2声明成员函数show为虚函数只是在其函数原型之前加上了关键字virtual。

【例15-2】 编写具有继承关系的两个类，在基类中定义一个虚函数，并在派生类中重定义该函数，然后用基类指针操作基类和派生类的对象，并调用该成员函数，检查该调用过程是否显示多态性。

```
//文件 media.h: 定义类 Media
#if !defined __MEDIA__H__
#define __MEDIA__H__

class Media
{
public:
    //构造函数
    Media(char * n,char * c);
    //析构函数
    ~Media();

    virtual void showInfo();

private:
    char * name;                //媒体名
    char * company;             //制作公司
};
```

```
#endif
//文件 media.cpp: 类 Media 的实现
#include <string.h>
#include <iostream.h>
#include "media.h"
//构造函数的实现
Media::Media(char * n,char * c)
{
    name=new char[strlen(n)+1];
    strcpy(name,n);
    company=new char[strlen(c)+1];
    strcpy(company,c);
}
//析构函数的实现
Media::~Media()
{
    if (name!=NULL)
        delete []name;
    if (company!=NULL)
        delete []company;
}
void Media::showInfo()
{
    cout<<"Name: "<<name<<endl;
    cout<<"Company: "<<company<<endl;
}
//文件 audioMedia.h: 定义类 AudioMedia
#if !defined __AUDIOMEDIA__H__
#define __AUDIOMEDIA__H__
#include "media.h"
class AudioMedia: public Media
{
public:
    AudioMedia(char * n,char * c,char * s);
    ~AudioMedia();
    void showInfo();
private:
    char * singer;                              //歌手
};
#endif
//文件 audioMedia.cpp: 类 AudioMedia 的实现
#include <string.h>
```

```
#include <iostream.h>
#include "audioMedia.h"

AudioMedia::AudioMedia(char * n,char * c,char * s)
:Media(n,c)                                    //调用基类的构造函数
{
    singer=new char[strlen(s)+1];
    strcpy(singer,s);
}

AudioMedia::~AudioMedia()
{
    if (singer!=NULL)
        delete []singer;
}

void AudioMedia::showInfo()
{
    Media::showInfo();                         //调用基类被重定义的函数
    cout<<"Singer: "<<singer<<endl;
}
```

```
//文件 ex15_2.cpp:测试使用基类指针操作继承关系中的各个对象
#include <iostream.h>
#include "media.h"
#include "audioMedia.h"

int main()
{
    Media * mPtr;
    Media medium("Slumdog Millionaire","Celador Films");
    AudioMedia audio("The Colour of My Love","Columbia","Celine Dion");

    mPtr=&medium;                              //基类指针指向基类对象
    mPtr->showInfo();                          //用基类指针调用成员函数,动态绑定

    cout<<endl;

    mPtr=&audio;                               //基类指针指向派生类对象
    mPtr->showInfo();                          //用基类指针调用成员函数,动态绑定

    return 0;
}
```

运行结果:

```
Name: Slumdog Millionaire
Company: Celador Films

Name: The Colour of My Love
Company: Columbia
Singer: Celine Dion
```

例15-2中类AudioMedia的定义和例15-1中定义的完全相同,但由于类AudioMedia继承自类Media,而类Media的成员函数showInfo被声明为虚函数之后,其派生类AudioMedia的成员函数showInfo也是虚函数,虽然其函数声明前面没有加关键字virtual。例15-2的main函数和例15-1中的main函数也完全相同。例15-2先使用基类指针mPtr指向基类对象medium,并调用成员函数showInfo,由于showInfo是虚函数,引起动态绑定,因为接收消息的对象是medium,其类型为Media,所调用的函数就是定义于类Media中的showInfo函数,因此语句

```
mPtr->showInfo();
```

所产生的输出结果是:

```
Name: Slumdog Millionaire
Company: Celador Films
```

当指针mPtr指向派生类对象audio并调用函数showInfo时,仍然会引起动态绑定,此时接收消息的对象是audio,其类型为AudioMedia,语句

```
mPtr->showInfo();
```

实际执行的函数就是定义于派生类AudioMedia中的成员函数showInfo,产生的输出结果就是:

```
Name: The Colour of My Love
Company: Columbia
Singer: Celine Dion
```

可以看到,同一条函数调用语句

```
mPtr->showInfo();
```

由于使用基类指针调用虚函数,程序运行时会根据接收消息的对象的类型来确定其所调用的函数,于是就在不同的情况下调用了不同的函数,产生了不同的结果,这就是多态性。

15.3 抽象基类和纯虚函数

有了虚函数,就可以利用多态性编写出更加灵活的程序。多态性要求程序在基类中将相应的成员函数(引发多态性的函数)声明为虚函数,有时这种方式并不合适。例如,假设三角形类Triangle、圆形类Circle、矩形类Rectangle都是形状类Shape的派生类,在类Triangle中有函数area计算三角形的面积,类Circle和类Rectangle也分别有自己的函数area计算圆形和矩形的面积(见图15-1),为了更好地操作这些类的对象,希望利用多态性的机制,这就需要将函数area在基类Shape中声明为虚函数。但如何实现基类Shape中的area函数呢?类Shape的对象是什么样子?它的面积该怎么计算?因为在这个问题中,基类Shape仅仅是一个抽象的概念,它不存在具体的对象实例。所以,无法具体描述一个Shape对象的属性,当然也更无法描述一个Shape对象面积的计算方法。

如果有一个抽象的类(就像Shape),它没有实际的实例化对象(或其对象的属性无法描

述)，该类的定义仅是为派生类提供一个框架(或接口)，也不用实现它，最好也不允许程序员声明该类的对象，这样就可以很好地解决上面的问题了。纯虚函数可以帮助解决这类问题。

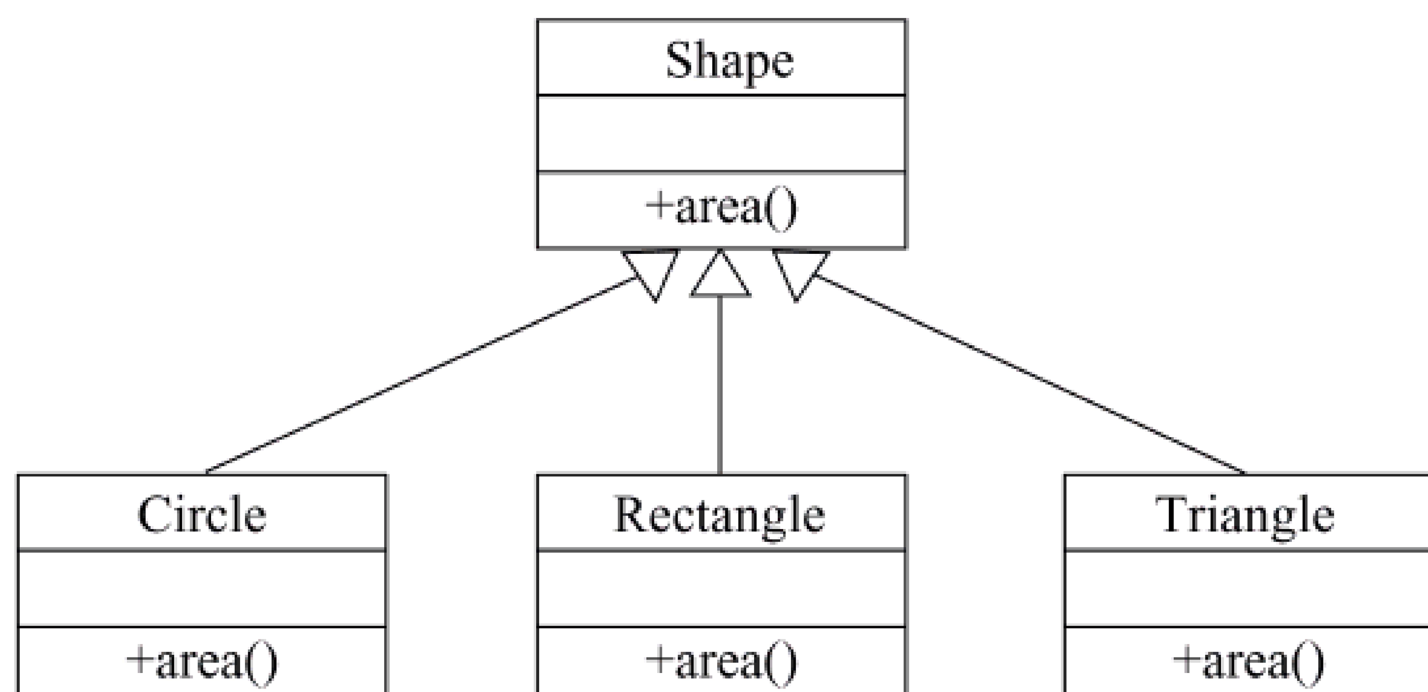


图 15-1　形状的继承结构

15.3.1　纯虚函数

纯虚函数是没有函数体的虚函数，纯虚函数在声明时要“初始化为 0”。例如，将类 Shape 中的函数 area 声明为纯虚函数的语法为：

```
virtual double area()=0;              //纯虚函数
```

一个类可以包含一个或多个纯虚函数，包含纯虚函数的类又称为抽象类，抽象类是指没有实例化对象的类。如果一个类是从一个包含纯虚函数的类中派生而来，并且在该派生类中没有提供所有纯虚函数的实现，那么该派生类也是一个抽象类。

抽象类是定义不完全的类，不能实例化对象。试图创建一个抽象类的对象时，编译器会报错。

15.3.2　抽象类和具体类

C++ 中定义的类是一种新的数据类型，一般认为，这些新类型是可以用来声明变量(实例化的对象)的。纯虚函数允许定义不能实例化任何对象的类，即抽象类。抽象类存在的意义在于它们将作为基类被其他类继承，所以抽象类有时也称为“抽象基类”。

抽象类不能用来创建实例化对象，它唯一的用途就是为其他类提供合适的基类。具体地说，抽象类可以为一些相关的类提供共同的框架(接口)，可以用继承的关系将这些类组织成树形结构。相对抽象类而言，可以建立实例化对象的类就是具体类。具体类中不再包含未被实现的纯虚函数。抽象类中的虚函数(包括纯虚函数)就构成了这些从它派生出来的具体类的共同接口。

在下面的例子中，类 Circle 和类 Rectangle 都是类 Shape 的派生类。在类 Circle 和类 Rectangle 中都定义了函数 show 和 area，为了能更好地操作这些类的对象(使用基类指针利用多态性)，程序在基类 Shape 中将函数 show 和 area 声明为虚函数。由于类 Shape 的成员函数 area 无法实现，而且程序也不需要创建类 Shape 的实例化对象，因此程序将函数 show 和 area 都声明为纯虚函数，类 Shape 就是一个抽象基类。

下面例子的程序分为 5 个部分。其中，第一部分(例 15-3)是抽象基类 Shape 的定义。类 Shape 的定义中只包含两个公有成员函数 area 和 show，它们都被“初始化为 0”，即都是

纯虚函数。纯虚函数不需要给出函数的实现,所以抽象基类 Shape 没有实现部分。另外,由于 area 和 show 函数都不需要修改对象的数据成员,所以进一步将它们声明为 const 成员函数。

【例 15-3】 抽象基类 Shape 的定义。

```
//文件 shape.h: 定义抽象基类 Shape
#if !defined __SHAPE__H__
#define __SHAPE__H__

class Shape {
public:
    virtual double area() const=0;
    virtual void show() const=0;
};

#endif
```

第二部分(例 15-4)和第三部分(例 15-5)是派生类 Circle 的定义和实现。类 Circle 重新定义并实现了函数 show 和 area,虽然类 Circle 的成员函数 show 和 area 的函数原型前面没有关键字 virtual,但因为它们在基类 Shape 中都被定义为虚函数,所以在类 Circle 中它们仍然是虚函数。类 Circle 实现了基类中定义的所有纯虚函数,它不再包含纯虚函数,所以类 Circle 是具体类。

【例 15-4】 派生类 Circle 的定义。

```
//文件 circle.h: 定义派生类 Circle
#if !defined __CIRCLE__H__
#define __CIRCLE__H__

#include "shape.h"

class Circle: public Shape
{
public:
    Circle(double=0.0,double=0.0,double=1.0);
    double area() const;
    void show() const;

private:
    double x,y;
    double r;
};

#endif
```

【例 15-5】 派生类 Circle 的实现。

```
//文件 circle.cpp: 实现类 Circle
#include "circle.h"
#include <iostream.h>
```

```
#define PI 3.1416
Circle::Circle(double a,double b,double c)
{
    x=a;
    y=b;
    r=c;
}
double Circle::area()const
{
    return PI*r*r;
}
void Circle::show()const
{
    cout<<"I am a Circle.";
}
```

第四部分(例15-6)和第五部分(例15-7)是派生类Rectangle的定义和实现。同样,类Rectangle重新定义并实现了函数show和area,它不再是一个抽象类,而是一个具体类。

【例15-6】 派生类Rectangle的定义。

```
//文件Rectangle.h:定义派生类Rectangle
#if !defined__RECTANGLE__H__
#define__RECTANGLE__H__
#include "shape.h"
class Rectangle: public Shape
{
public:
    Rectangle(double=1.0,double=1.0);
    double area() const;
    void show() const;
private:
    double length;
    double width;
};
#endif
```

【例15-7】 派生类Rectangle的实现。

```
//文件rectangle.cpp:实现类Rectangle
#include "Rectangle.h"
#include <iostream.h>
Rectangle::Rectangle(double a,double b)
{
    length=a;
```

```
        width=b;
    }
    double Rectangle::area() const
    {
        return length* width;
    }
    void Rectangle::show() const
    {
        cout<<"I am a Rectangle.";
    }
```

例15-8对本节上述程序中的3个类进行了测试。例15-8中定义了一个函数callArea，它通过引用参数接受一个Shape类的对象，当然它也能接受Shape类的任何派生类的对象(因为派生类对象同时也是基类的对象)。在main函数中首先声明了一个Circle类型的对象cir和一个Rectangle类型的对象rec(类Shape是抽象类，不能实例化对象)，然后调用函数callArea。第一次调用callArea函数时，实参是对象cir，由于callArea采用传引用调用方式，形参obj实际上是对Circle对象cir的引用，用引用变量obj调用虚函数show时采用动态绑定的机制，所以这里被调用的函数不是定义于基类中的函数，而是变量obj实际引用的对象所属的类中定义的函数，即定义于类Circle中的函数show。同样，函数调用obj. area()也体现了多态性特征，实际调用的函数也是定义于类Circle中的函数area。

main函数第二次调用函数callArea时，实参是类Rectangle的对象rec，函数callArea中的两次函数调用obj. show()和obj. area()实际调用的都是定义于类Rectangle中的show和area函数。

【例15-8】 编写程序使用本节上述程序(例15-3至例15-7)中定义的从抽象基类派生出来的两个具体类：类Circle和类Rectangle，并编写统一的程序框架来操作它们的实例化对象。

```
//文件 ex15_8.cpp: 对类 Shape 的层次结构进行多态性测试
#include <iostream.h>
#include "Circle.h"
#include "Rectangle.h"
void callArea(Shape &);
int main()
{
    Circle cir(0.0,0.0,2.5);
    Rectangle rec(2.4,5.3);
    callArea(cir);
    callArea(rec);
    return 0;
}
void callArea(Shape &obj)
```

```
{
    obj.show();
    cout<<" My area is: "<<obj.area()<<endl;
}
```

运行结果：

```
I am a Circle. My area is: 19.635
I am a Rectangle. My area is: 12.72
```

一个类的层次结构中可以不包含任何抽象类。但有时在类层次结构的顶部使用抽象类对程序的结构和控制是有很大帮助的。

例如，上面的 callArea 函数就是一个只与接口(抽象类 Shape)有关、而与实现该接口的具体类(Circle 和 Rectangle)无关的程序模块。具体类 Circle 和 Rectangle 可以有自己对接口的实现方式，而这种实现方式与类的使用者 callArea 无关。这样的好处是显著提高了程序的可维护性和可扩展性：如果 Circle 或 Rectangle 修改自己的 show 或 area 实现，callArea 无须修改就能使用新的版本；如果要新增一个类 Shape 的派生具体类，成员函数 callArea 可以无缝地处理该类的对象。

程序设计方法提示：利用多态性，可以根据接口设计出与实现无关的程序。

*15.4 虚析构函数

类的构造函数不能是虚函数，但析构函数可以是虚函数，而且在很多时候，析构函数最好声明为虚函数。

构造函数的作用是初始化类的对象的数据成员，它首先调用继承关系中最顶层的构造函数，然后按继承的顺序从上往下依次调用各派生类的构造函数的代码。析构函数则正好相反，它首先执行自己的析构函数的代码，然后沿着继承的层次从下往上依次调用各基类的析构函数。通常情况下，析构函数的这种调用关系是没有问题的。

如果程序需要用指针操作类层次结构中动态生成的对象时会怎么样呢？用 new 运算符动态创建派生类对象时没有什么问题，new 运算符是根据跟着它后面的操作数指明的类型去创建相应的对象，其语义是明确的，创建好的对象可以交给各层基类型的指针。但是，要用 delete 运算符撤销动态创建的对象时就会有问题，如果用 delete 运算符作用于一个类指针，则不管该指针当时实际指向什么类型的对象，该操作调用的析构函数始终是声明该指针时所使用的类的析构函数，而不是实际所指的对象的所属类的析构函数。因此，当指针实际指向的对象不属于指针声明的类时，这种操作方式就可能不能正确地撤销对象。而在面向对象编程中，使用基类指针操作派生类对象是非常普遍的。

多态性可以帮助解决这个问题。一个简单方法就是将基类的析构函数声明为虚函数。这样，用 delete 撤销一个基类指针所指的对象时，就能根据对象的实际类型调用相应的析构函数。声明虚析构函数的语法是在析构函数的函数原型前面加上 virtual 关键字。如声明类 Test 的析构函数为虚函数的语句为：

```
virtual ~Test();
```

将基类的析构函数声明为虚函数,可以使所有派生类的析构函数自动成为虚函数(即使派生类的析构函数和基类的析构函数不同名),而且各派生类的析构函数的函数原型前面也可以不加 virtual 关键字。这时,如果使用基类指针操作各派生类的对象,并用 delete 运算符撤销基类指针所指的对象时,就会引发动态绑定,程序不会根据指针的类型来选择执行哪个类的析构函数,而是根据指针所指的具体对象的实际类型来确定所要执行的析构函数,这样就可以正确地撤销动态对象了。

下面是利用基类指针动态创建和撤销类对象的例子,其中使用了虚析构函数。

【例 15-9】 编写一个管理企业雇员信息的简单程序,并利用基类指针创建和撤销所有雇员对象。

```
//文件 employee.h: 定义基类 Employee
#if !defined __EMPLOYEE__H__
#define __EMPLOYEE__H__
#include <iostream.h>
class Employee
{
public:
    Employee()
    {
        cout<<"Employee begin!"<<endl;
    }
    virtual ~Employee()
    {
        cout<<"Employee end!"<<endl;
    }
};
#endif
//文件 programmer.h: 定义基类 Programmer
#if !defined __PROGRAMMER__H__
#define __PROGRAMMER__H__
#include <iostream.h>
#include <string.h>
#include "Employee.h"
class Programmer: public Employee
{
public:
    Programmer(char * str)
    {
        cout<<"Programmer begin!"<<endl;
        name=new char[strlen(str)+1];
        strcpy(name,str);
    }
```

```
    ~Programmer()
    {
        if (name!=NULL)
            delete [] name;
        cout<<"Programmer end!"<<endl;
    }
private:
    char * name;
};
#endif
//文件 accountant.h: 定义基类 Accountant
#if !defined __ACCOUNTANT__H__
#define __ACCOUNTANT__H__
#include <iostream.h>
#include "Employee.h"
class Accountant: public Employee
{
public:
    Accountant(int n)
    {
        cout<<"Accountant begin!"<<endl;
        age=n;
    }
    ~Accountant()
    {
        cout<<"Accountant end!"<<endl;
    }
private:
    int age;
};
#endif
//文件 ex15_9.cpp
#include "accountant.h"
#include "programmer.h"
#define MAX_STR_LEN 256
#define MAX_EMP_NUM 100
int main()
{
    char name[MAX_STR_LEN];
    int age;
    int option;
```

```
//声明储存雇员信息的数组
Employee * emp[MAX_EMP_NUM], * empPtr;
int empNum=0;
for (int i=0;i<MAX_EMP_NUM;i++)
    emp[i]=NULL;
//输入雇员信息
cout<<"Input employees' info:"<<endl;
cout<<"1 --- Programmer"<<endl;
cout<<"2 --- Accountant"<<endl;
cout<<"0 --- exit"<<endl;
cin>>option;
while (option)
{
    switch (option)
    {
        case 1:
            //输入程序员信息
            cout<<"Please input his or her name: ";
            cin>>name;
            //创建程序员对象
            empPtr=new Programmer(name);
            emp[empNum++]=empPtr;
            break;
        case 2:
            //输入会计信息
            cout<<"Please input his or her age: ";
            cin>>age;
            //创建会计对象
            empPtr=new Accountant(age);
            emp[empNum++]=empPtr;
            break;
        default:
            break;
    }
    cout<<endl<<"Input employees' info:"<<endl;
    cout<<"1 --- Programmer"<<endl;
    cout<<"2 --- Accountant"<<endl;
    cout<<"0 --- exit"<<endl;
    cin>>option;
}
//撤销所有雇员对象
for (i=0;i<empNum;i++)
    delete emp[i];
```

```
    return 0;
}
```

运行结果：

```
Input employees' info:
1 --- Programmer
2 --- Accountant
0 --- exit
1
Please input his or her name: Zhang
Employee begin!
Programmer begin!
Input employees' info:
1 --- Programmer
2 --- Accountant
0 --- exit
2
Please input his or her age: 28
Employee begin!
Accountant begin!
Input employees' info:
1 --- Programmer
2 --- Accountant
0 --- exit
0
Programmer end!
Employee end!
Accountant end!
Employee end!
```

例 15-9 中的程序定义了基类 Employee，并将其析构函数声明为虚析构函数，然后定义了派生类 Programmer 和 Accountant。两个派生类分别需要不同的数据(名字或年龄)来初始化自己的实例化对象，并且 3 个类的构造函数和析构函数中各有一条输出语句以输出该函数被调用的信息。

main 函数首先声明了一个基类指针数组 emp 和一个基类的临时指针 empPtr，然后通过一个 while 循环输入各雇员的具体信息，在输入过程中通过雇员类型动态创建不同派生类的实例化对象，并放到数组 emp 中进行统一管理。用 new 运算符创建不同类型的对象时需要明确指出要创建对象的类型。例如，语句

```
empPtr=new Programmer(name);
```

创建一个派生类 Programmer 的对象。而语句

```
empPtr=new Accountant(age);
```

则创建了派生类 Accountant 的一个动态对象。

程序最后通过一个 for 循环使用 delete 运算符撤销所有创建的对象,撤销对象的语句都是:

```
delete emp[i];
```

从程序的运行结果可以看出,撤销第一个对象(Programmer 对象)时,程序先调用了类 Programmer 的析构函数,然后调用基类 Employee 的析构函数,保证了动态对象被正确地撤销。撤销第二个对象(Accountant 对象)时,则是先调用派生类 Accountant 的析构函数,然后调用基类 Employee 的析构函数。

这个例子说明:如果将析构函数声明为虚析构函数,用 delete 运算符撤销动态对象时,根据多态的特性,程序不是根据指针的类型决定调用哪个类的析构函数,而是根据指针所指向的对象的实际类型来调用相应类的析构函数。

虚析构函数的这种特性,为编写使用基类指针操作继承层次中各个类的动态对象带来极大的方便。

15.5 软件渐增式开发

第 14 章讨论了利用继承和复合的方法进行软件的渐增式开发。可以先开发功能较简单的类并完成对该类对象的操作,然后在此基础上,通过复合和继承的方法建立新的功能强大的类,这样可很容易地将新类的对象纳入已有的程序框架中。虚函数和多态性的特性可以使软件的渐增式开发更加方便和自然。

本节介绍一个利用多态性实现软件渐增式开发的例子。这是一个简单的媒体资料管理系统,是使用继承和多态性对例 12-2 的改进。为了能使用面向对象的方法管理各种媒体资料的信息,程序先编写了一个抽象基类 Media,定义了所有音像资料的公共接口,并编写了资料架类 shelf,实现了对各种音像资料的统一管理。

系统实现的第一步是管理音频媒体的信息。从类 Media 派生出音频媒体类 AudioMedia,类 AudioMedia 包含了音频媒体的基本信息并实现了 showInfo 函数,是一个具体类。接下来编写了主控程序,实现了通过类 Shelf 对类 AudioMedia 的对象的管理。

系统实现的第二步是增加对视频媒体信息的管理,在前面程序的基础上增加了一个新类 VideoMedia。类 VideoMedia 也是从抽象基类 Media 派生而来,是一个具体类,包含了视频媒体的基本信息并实现了 showInfo 函数。在这一步也编写了测试程序,该测试程序的主控模块和第一步用的主控模块完全一样,不用对主控模块进行任何修改就可以将新类 VideoMedia 的对象纳入管理。

系统实现的第三步是增加对软件资料信息的管理,在前面程序的基础上增加了新类 Software,类 Software 同样继承自抽象基类 Media,并包含了软件资料的基本信息,实现了自己的 showInfo 函数。系统的主控程序仍然不需要进行任何修改就可以管理类 Software 的对象。

图 15-2 描述了各类之间的关系。

本节依次给出分三步实现一个系统的例子。第一步由例 15-10~例 15-15(6 个部分)以及例 15-16 组成,例 15-10 和例 15-11 是抽象类 Media 的定义和实现。类 Media 包含一个纯

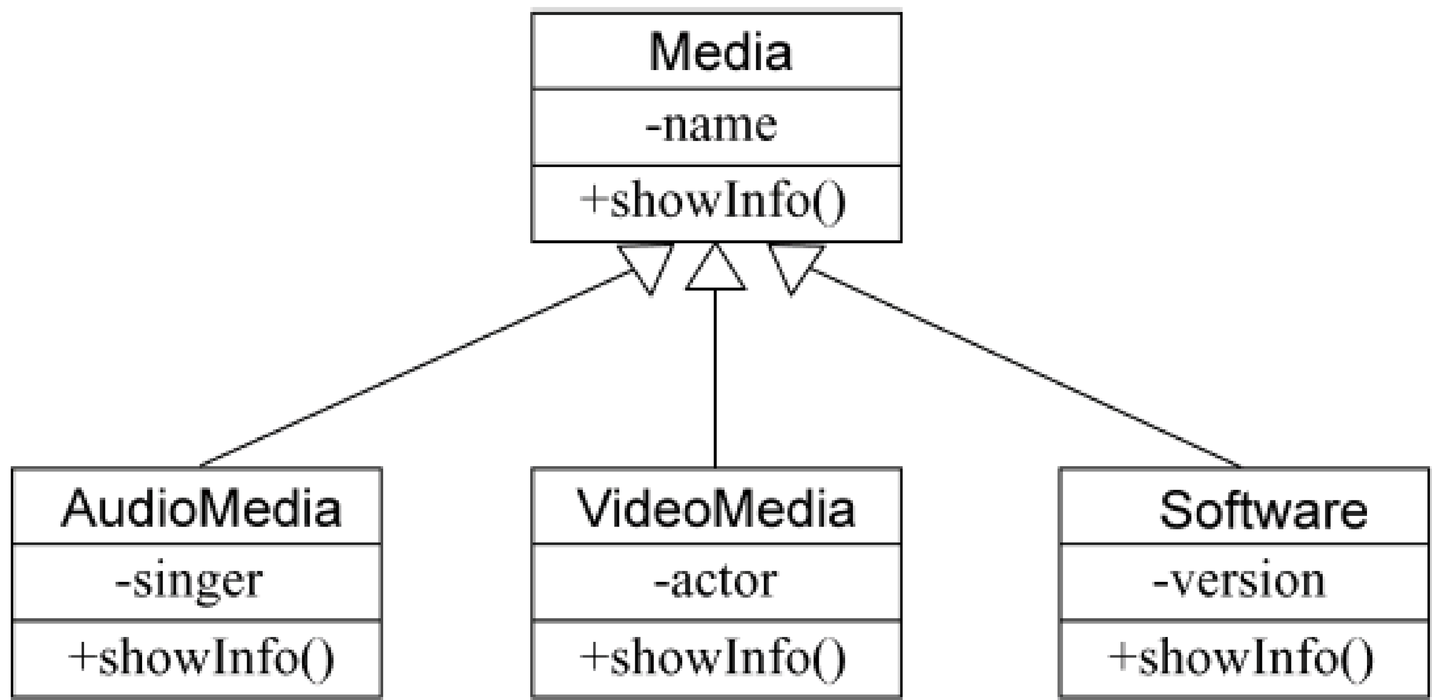


图 15-2 各类之间的关系

虚函数 showInfo,数据成员 name 被定义成受保护成员,以便继承。

【例 15-10】 抽象基类 Media 的定义。

```
//文件 media.h: 定义类 Media
#if !defined __MEDIA__H__
#define __MEDIA__H__
class Media
{
public:
    //构造函数
    Media(char * n);
    //析构函数
    ~Media();
    virtual void showInfo()=0;
protected:
    char * name;                    //媒体名
};
#endif
```

【例 15-11】 抽象基类 Media 的实现。

```
//文件 media.cpp: 类 Media的实现
#include <string.h>
#include <iostream.h>
#include "media.h"
//构造函数的实现
Media::Media(char * n)
{
    name=new char[strlen(n)+1];
    strcpy(name,n);
}
//析构函数的实现
Media::~Media()
{
```

```
    if (name!=NULL)
        delete []name;
}
```


例15-12和例15-13是派生类AudioMedia的定义和实现。该类增加一个数据成员singer,并重新定义成员函数showInfo,输出其对象的类型、名称和歌手信息。派生类AudioMedia实现了纯虚函数,不是抽象类。

【例15-12】 派生类AudioMedia的定义。

```
//文件audioMedia.h:定义类AudioMedia
#if !defined __AUDIOMEDIA__H__
#define __AUDIOMEDIA__H__
#include "media.h"

class AudioMedia: public Media
{
public:
    AudioMedia(char * n,char * s);
    ~AudioMedia();

    void showInfo();
private:
    char * singer;                          //歌手
};

#endif
```

【例15-13】 派生类AudioMedia的实现。

```
//文件audioMedia.cpp:类AudioMedia的实现
#include <string.h>
#include <iostream.h>
#include "audioMedia.h"

AudioMedia::AudioMedia(char * n,char * s)
: Media(n)                                  //调用基类的构造函数
{
    singer=new char[strlen(s)+1];
    strcpy(singer,s);
}

AudioMedia::~AudioMedia()
{
    if (singer!=NULL)
        delete []singer;
}

void AudioMedia::showInfo()
{
    cout<<"AudioMedia,"<<name<<","<<singer<<endl;
}
```

例 15-14 和例 15-15 是用于管理媒体的资料架类 Shelf 的定义和实现。类 Shelf 定义了一个 Media 类型的指针数组 media，最多能存放管理 50 个媒体对象，实现了 add 成员函数用于将媒体对象增加到资料架中，实现了 list 成员函数输出资料架中所有媒体对象的信息。

【例 15-14】 资料管理类 Shelf 的定义。

```
//文件 shelf.h: 定义复合类 Shelf
#if !defined __SHELF__H__
#define __SHELF__H__

#include "media.h"
#define MAX_MEDIA_NUM    50

class Shelf {
public:
    Shelf();
    int add(Media * m);             //资料架增加一个媒体资料
    void list();                    //列出资料架上所有的媒体资料

private:
    Media * media[MAX_MEDIA_NUM];
    int num;
};

#endif
```

【例 15-15】 资料管理类 Shelf 的实现。

```
//文件 shelf.cpp: 类 Shelf 的实现部分
#include <iostream.h>
#include "shelf.h"

Shelf::Shelf()
{
    num=0;
}

//资料架增加一个媒体资料
int Shelf::add(Media * m)
{
    if (num >=MAX_MEDIA_NUM)
        return 0;

    media[num]=m;
    num++;

    return 1;
}

//列出资料架上所有的媒体资料
void Shelf::list()
{
```

```
        int i;
        for (i=0;i<num;i++)
        {
            cout<<"No."<<i+1<<": ";
            media[i]->showInfo();
        }
}
```

例15-16测试了本节上面诸程序中定义的类,其中的main函数首先声明了类Shelf的一个对象shelf和类AudioMedia的对象audio,并通过语句

```
shelf.add(&audio);
```

将对象audio加入shelf中。然后,调用shelf的成员函数list输出资料架中的媒体信息。成员函数list使用for循环遍历其中的所有媒体,并通过函数调用:

```
media[i]->showInfo();
```

输出相应媒体的具体信息。在输出具体信息时,采用的是基类指针加虚函数的调用方式,引起动态绑定,所以该语句实际上调用的是指针所指对象的相应成员函数。这里实际调用的是派生类AudioMedia的成员函数showInfo。

【例15-16】 编写程序测试本节上述程序中定义的类,通过类Shelf实现对媒体对象的管理。

```
//文件ex15_16.cpp:利用多态性,
//通过类Shelf实现对AudioMedia媒体对象的管理
#include <iostream.h>
#include "shelf.h"
#include "audioMedia.h"
int main()
{
    Shelf shelf;
    AudioMedia audio("The Colour of My Love","Celine Dion");
    shelf.add(&audio);
    shelf.list();
    return 0;
}
```

运行结果:

```
No.1: AudioMedia,The Colour of My Love,Celine Dion
```

系统实现的第二步是增加对视频媒体的管理。下面程序定义并实现了派生类VideoMedia,继承于例15-10中所定义的抽象类Media。派生类VideoMedia增加了私有成员actor,并实现基类的虚函数showInfo,以输出该类对象的类型、名称和主要演员信息。类VideoMedia不含纯虚函数,是具体类。

【例15-17】 派生类VideoMedia的定义。

```
//文件 videoMedia.h: 定义类 VideoMedia
#if !defined __VIDEOMEDIA__H__
#define __VIDEOMEDIA__H__
#include "media.h"

class VideoMedia: public Media
{
public:
    VideoMedia(char * n,char * a);
    ~VideoMedia();

    void showInfo();

private:
    char * actor;                                  //演员
};

#endif
```

【例 15-18】　派生类 VideoMedia 的实现。

```
//文件 videoMedia.cpp: 类 VideoMedia 的实现
#include <string.h>
#include <iostream.h>
#include "videoMedia.h"

VideoMedia::VideoMedia(char * n,char * a)
:Media(n)                                          //调用基类的构造函数
{
    actor=new char[strlen(a)+1];
    strcpy(actor,a);
}

VideoMedia::~VideoMedia()
{
    if (actor!=NULL)
        delete []actor;
}

void VideoMedia::showInfo()
{
    cout<<"VideoMedia,"<<name<<","<<actor<<endl;
}
```

例 15-19 对例 15-16 中的程序稍作修改，实现了对音频和视频媒体的统一管理。和例 15-6 中的程序相比，例 15-19 仅增加了两条语句，声明了类 VideoMedia 的一个对象 video，并将它加入到对象 shelf 中。对象 shelf 将会对其列表中的所有媒体对象逐个输出，在输出对象 video 的详细信息时，语句

```
media[i]->showInfo();
```

调用的是定义于类 VideoMedia 中的 showInfo 函数。

【例15-19】 修改例15-16中的程序，增加例15-17中定义的类VideoMedia，实现对音频和视频媒体的统一管理。

```
//文件 ex15_19.cpp: 利用多态性，
//通过类 Shelf 实现对 AudioMedia 和 VideoMedia 媒体对象的管理
#include <iostream.h>
#include "shelf.h"
#include "audioMedia.h"
#include "videoMedia.h"

int main()
{
    Shelf shelf;
    AudioMedia audio("The Colour of My Love","Celine Dion");
    VideoMedia video("Forrest Gump","Tom Hanks");

    shelf.add(&audio);
    shelf.add(&video);
    shelf.list();

    return 0;
}
```

运行结果：

```
No.1: AudioMedia,The Colour of My Love,Celine Dion
No.2: VideoMedia,Forrest Gump,Tom Hanks
```

系统实现的第三步是增加对软件资料媒体的管理。例15-20和例15-21与例15-17和例15-18类似，定义并实现了抽象类Media的一个派生类Software。类Software增加了私有成员版本号version，并实现了虚函数show，它输出该类对象的类型、名称和版本号信息。类Software是一个具体类。例15-7是测试程序，和例15-6中的程序相比，也只增加了两条语句，声明了类Software的一个对象vc，并将该对象加入到one中。类Shelf不需要做任何修改就可以将新加入的类Software的对象纳入管理。

【例15-20】 派生类Software的定义。

```
//文件 software.h: 定义类 Software

#if !defined __SOFTWARE__H__
#define __SOFTWARE__H__
#include "media.h"

class Software: public Media
{
public:
    Software(char * n,char * v);
    ~Software();

    void showInfo();

private:
```

```
    char * version; //版本
};
#endif
```

【例 15-21】 派生类 Software 的实现。

```
//文件 software.cpp: 类 SoftWare 的实现
#include <string.h>
#include <iostream.h>
#include "software.h"
Software::Software(char * n,char * v)
: Media(n)                              //调用基类的构造函数
{
    version= new char[strlen(v)+1];
    strcpy(version,v);
}
Software::~ Software()
{
    if (version!=NULL)
        delete []version;
}
void Software::showInfo()
{
    cout<<"Software,"<<name<<","<<version<<endl;
}
```

【例 15-22】 修改例 15-19 中的程序，利用同样的方式，实现对音频媒体、视频媒体和软件资料的统一管理。

```
//文件 ex15_6.cpp: 利用多态性,
//通过类 Shelf 实现对 AudioMedia、VideoMedia 和 Software 媒体对象的管理
#include <iostream.h>
#include "shelf.h"
#include "audioMedia.h"
#include "videoMedia.h"
#include "software.h"
int main()
{
    Shelf shelf;
    AudioMedia audio("The Colour of My Love","Celine Dion");
    VideoMedia video("Forrest Gump","Tom Hanks");
    Software vc("Visual C++","6.0");
    shelf.add(&audio);
    shelf.add(&video);
    shelf.add(&vc);
```

```
    shelf.list();

    return 0;
}
```

运行结果:

```
No.1: AudioMedia,The Colour of My Love,Celine Dion
No.2: VideoMedia,Forrest Gump,Tom Hanks
No.3: Software,Visual C++,6.0
```

分析上面分三步逐步实现一个系统的例子,不难发现,如果利用多态性进行软件的渐增式开发,往往只需要在基类中定义好接口(公共的虚函数),就可以根据基类的接口编写这些类的对象的处理程序,而不必关心每个派生类的具体实现。采取这种方法,即使在软件开发过程中需要增加新的派生类,只要它依然遵循基类定义的接口,所有按照接口使用这些对象的程序都不用修改,就可直接处理这些新增的对象了。这要比上一章所讨论的只使用复合和继承来进行软件的渐增式开发更加灵活。

习　题　15

15.1　填空:

(1) 如果一个类包含一个或多个纯虚函数,则该类为________。

(2) 抽象基类________实例化对象,________被继承,________声明类指针。

(3) 类的多态性是通过________实现的。

(4) 派生类继承基类的虚函数后________不重新定义该函数。

(5) 在派生类中重新定义虚函数时,必须在________和________上和基类保持一致。

(6) 通过一个对象调用虚函数时,C++系统对该调用采用________。

15.2　什么是C++的多态性?它是通过什么函数来实现的?

15.3　什么是虚函数?纯虚函数有什么作用?

15.4　虚析构函数的作用是什么?举一个适合使用虚析构函数的例子。

15.5　有一个汽车类Vehicle,将它作为基类派生出小车类car和卡车类truck,定义这些类,并定义一个虚函数用来显示各类的信息。

15.6　以15.5节例子中的程序为基础,建立一个内容相对丰富的媒体信息管理系统:

(1) 丰富音频媒体类、视频媒体类、软件类的内容,使这些类能描述尽可能多的和该媒体相关的信息。

(2) 在3个派生类的基础上丰富基类Media的内容,将3个派生类中的所有公共信息提到基类中去。

(3) 在类Shelf中采用链表管理各种媒体,并实现媒体对象的插入、归类、删除、查询以及信息的输入输出。

(4) 编写程序测试上述所有类,形成一个可运行的管理系统。

第16章 文件和流

【学习内容】

本章介绍C++的文件和流。主要内容包括：

- ◆ 文件和流的概念。
- ◆ 使用FILE结构打开、建立、读和写文件。
- ◆ 使用文件流打开、建立、读和写文件。

【学习目标】

- ◆ 掌握文件和流的基本概念。
- ◆ 掌握使用FILE结构和fopen函数建立和打开文件。
- ◆ 掌握使用FILE结构读写文本文件。
- ◆ 掌握使用FILE结构读写二进制文件。
- ◆ 了解文件流类的继承关系。
- ◆ 掌握使用ofstream类建立、打开和读文件。
- ◆ 掌握使用ifstream类打开和写文件。
- ◆ 掌握使用文件流读写文本文件。
- ◆ 掌握使用文件流读写二进制文件。

程序运行时存储在变量、数组和对象等数据结构中的数据都是临时的，只存在于程序运行过程当中，程序运行结束后这些数据都会随着程序被撤销而消失。如果想要把程序运行期间的数据保存下来，留待下次运行时使用，可以使用文件技术。本章将介绍文件和流的基本概念，并讨论如何通过FILE结构以及文件流的方式进行文件操作。

16.1 基本概念

文件是根据特定目的而收集在一起的相关数据的集合。C++把每个文件都看成是一个有序的字节流，每个文件都以文件结束标志(EOF)结束(见图16-1)。如果要操作某个文件，程序必须首先打开该文件。当一个文件被打开后，该文件就和一个流关联起来，这里的流实际上就是一个字节序列，是打开文件后操作系统为该文件建立的一个缓冲区。流是文件和程序之间通信的通道。这里的“文件”可以是一般意义上的存储在硬盘上的文件，也可以是具有输入输出功能的外设，如键盘、显示器等。当进行标准输入操作时，数据从键盘流

向程序,标准输出时数据则从程序流向显示器。

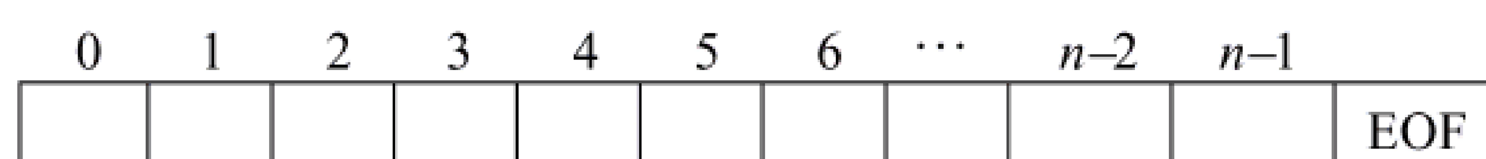


图 16-1 包含 n 个字节的文件

在 C++ 中,键盘、显示器和打印机等输入输出设备也被当作文件,也可以通过流对这些设备进行操作。所以,在本章看到的文件的输入输出与第 3 章介绍的内容有许多相似之处。只不过在第 3 章中所处理的“文件”是标准的输入输出设备,而本章主要介绍如何处理用户自己定义的文件(普通的磁盘文件)。

C++ 将普通文件分为文本文件和二进制文件。二进制文件一般含有特殊的格式或计算机代码,如图形文件和可执行文件等。文本文件则是可以用任何文字处理程序阅读和编辑的简单 ASCII 文件。二进制文件中包含的数据往往是一些看不懂的“乱码”,C++ 对包含特殊格式的二进制文件的操作方式一般都是按“块”操作,通过对“块”进行不同的解释来获取其中的信息。用 FILE 结构操作二进制文件时一般使用 fread 和 fwrite 函数,使用文件流操作二进制文件则一般使用 read 和 write 成员函数,但这并不意味着不能使用其他文件输入输出函数来操作二进制文件。对文本文件的操作则一般使用的是字符和字符串的输入输出方式,但也可以按“块”操作。C++ 并没有对二进制文件和文本文件进行严格的区分。

一个程序在执行时都会自动打开 3 个文件以及与这 3 个文件相关联的流——标准输入流、标准输出流和标准错误流。标准输入流使程序能够读取来自预定义的输入设备的数据,标准输出流可以帮助程序把数据输出到预定义的输出设备上,标准错误流则帮助程序将错误信息输出到预定义的设备上。

C 语言提供了标准的输入输出手段。如果程序包含了头文件<stdio.h>,就可以使用文件指针 stdin、stdout 和 stderr 来分别操作上面 3 个标准流,操作方式和用 FILE 结构定义出来的文件指针一样。stdin、stdout 和 stderr 是定义于头文件<stdio.h>中的 FILE 结构类型的指针,它们分别和标准输入流、标准输出流和标准错误流相关联。用户也可以定义自己的 FILE 结构类型的指针来操作特定的文件。

C++ 除了保持了 ANSI C 的输入输出函数以外,还提供了面向对象的输入输出手段。如果包含了头文件<iostream.h>,程序员就可以使用 3 个对象(cin、cout 和 cerr)来操作这 3 个流。使用这 3 个对象的方法就是给它们发送相应的消息(调用成员函数)。当然,也可以创建特定的对象来操作自己的文件。

16.2 通过 FILE 结构进行文件操作

FILE 结构是在头文件<stdio.h>中定义的,它包含用来处理文件的信息,如与操作系统的文件管理相关的文件描述子(file descriptor)等。头文件<stdio.h>中还提供了许多读写文件数据的函数。操作文件的函数的命名和操作标准输入输出流的函数的命名类似:函数 fgetc 和 fputc 与函数 getchar 和 putchar 类似,用于从文件中读出一个字符和向文件中写入一个字符;函数 fscanf 和 fprintf 的功能也与 scanf 和 printf 的功能类似,只是前者输入输出的设备不是标准输入输出流,而是特定的文件。本节将讨论如何使用 FILE 结构和函

数进行文件操作。

16.2.1 建立、打开和关闭文件

一个文件必须处于打开的状态才能进行读写操作。C++ 保留了 ANSI C 对文件的处理方式，程序对每一个文件都使用一个单独的 FILE 结构管理，每一个打开的文件都必须有一个单独声明的 FILE 类型的指针用来引用该文件。

定义于头文件<stdio.h>中的函数 fopen 可以用来建立一个新文件或者打开一个已存在的文件。该函数的原型为：

```
FILE * fopen(const char * filename,const char * mode);
```

其中，filename 是拟打开或新建的文件的路径和名字，mode 是一个字符串，说明了文件打开的方式。例如，字符串"r"表示打开一个供读取数据的文件；"w"表示建立或打开一个供写入数据的文件，如果该文件不存在，则创建并打开该文件，如果该文件已经存在，那么该文件内容将被废弃。表 16-1 列出了打开文件的各种方式。

表 16-1　函数 fopen 打开文件的方式

打开方式	描　　述
"r"	打开一个供读取数据的文件
"w"	建立或打开一个供写入数据的文件，如果该文件已经存在，则废弃文件内容
"a"	建立或打开一个供写入数据的文件，如果文件已经存在，则写入的数据将追加到文件的尾部
"r+"	打开一个已存在的文件，该文件可以写入和读出数据
"w+"	建立或打开一个可供读和写的文件，如果文件已存在则废弃文件内容
"a+"	建立或打开一个可供读和写的文件，如果文件已存在则写入的数据追加到文件的尾部
"b"	打开一个二进制文件

表 16-1 中列出的打开方式 b 只能和其他方式组合使用。例如，"wb"表示以二进制方式打开一个供写入数据的文件，"rb"表示以二进制方式打开一个供读取数据的文件。打开方式中包含字符"b"时表示程序要操作的是二进制文件，二进制文件允许输入输出无格式的二进制数据(内存块中的原始数据)。打开方式中不包含字符"b"时表示程序要操作的是文本文件，使用文本文件输入输出无格式的二进制数据时可能会出错。

注意：*以错误的方式打开文件时可能会破坏文件的内容。例如，不希望破坏文件内容时却使用了"w"打开方式，系统将会在不给出任何提示的情况下废弃文件中的所有内容。*

如果文件被正确地打开，函数 fopen 将返回一个指向 FILE 结构的指针，该指针指向的 FILE 结构管理了被打开的那个文件。在打开文件时如果发生错误(打开文件失败)，那么该函数将返回 NULL。可能的错误包括：

- 以读的方式打开一个不存在的文件。
- 打开一个无权访问的文件，如某些系统文件。
- 以写的方式打开文件时，磁盘空间不够用。

打开的文件在使用完之后，应当被关闭。关闭文件的函数原型是：

```
int fclose(FILE * stream);
```

该函数的功能是关闭FILE结构的指针stream所对应的被打开的文件。如果成功关闭,则fclose返回0值;否则,返回文件结束标志EOF,指示出错。fclose函数将文件缓冲中的数据写入磁盘中并切断文件和流之间的关联。如果文件操作完毕后没有使用fclose函数关闭文件,有可能会导致文件数据的丢失。

下面是一个打开文件的例子。

【例16-1】 编写程序使用fopen函数新建或打开一个文本文件,并写入字符串"This is a test file."。

```
//文件ex16_1.cpp:新建和打开文件
#include <stdio.h>

int main()
{
    //定义FILE指针
    FILE * fptr;
    //打开或创建文件
    if ((fptr=fopen("test.txt","w"))!=NULL)
    {
        //往文件中写入字符串
        fprintf(fptr,"This is a test file.");
        //关闭文件
        fclose(fptr);
    }
    else
        printf("Open file or create file error.\n");
    return 0;
}
```

程序运行后,在程序运行的当前目录多了一个文件test.txt,用文本处理软件打开该文件,发现其中只有一个字符串"This is a test file."。

例16-1中main函数的第一条语句声明了一个指向FILE结构的指针fptr。接下来,语句

```
fptr=fopen("test.txt","w");
```

调用函数fopen打开一个名为test.txt的文件,如果该文件不存在,则先创建该文件,并将返回值赋值给文件指针fptr。一个文件只有被打开后才和流关联,对文件或流进行读写操作需要通过文件指针进行(这里对文件"test.txt"的操作都需要通过fptr进行)。函数fopen带两个参数,其中第一个参数是要打开或创建的文件的路径和文件名,参数"test.txt"表示当前目录下的文件test.txt,当然也可以说明绝对路径,如"c:\\test\\test.txt"(注意字符串中表示路径中斜杠的方式)表示在C盘的test目录下的文件test.txt。第二个参数是文件打开的方式,参数"w"表示以写方式打开文件,如果该文件不存在,则先创建该文件然后打

开它；如果该文件存在，则打开该文件并废弃文件中所有内容。函数 fopen 返回被打开文件的 FILE 结构的首地址，或者在文件打开失败时返回空指针 NULL。

程序接下来测试函数 fopen 的返回结果，如果打开文件成功则使用语句

```
fprintf(fptr,"This is a test file.");
```

往文件中写入字符串"This is a test file."，否则使用 printf 在标准输出设备（显示器）上输出报错信息。函数 fprintf 和 printf 相似，都是输出格式化数据，只是函数 printf 将数据输出到屏幕上，而 fprintf 则输出到某个文件中，因而函数 fprintf 比 printf 要多带一个参数，它的第一个参数是一个文件指针，指明了要写入格式化数据的文件。

文件操作完毕之后程序调用 fclose 函数关闭该文件。

例 16-1 演示的是如何建立、打开和关闭文本文件，下一节的例 16-2 将演示如何操作一个二进制文件。

16.2.2 写文件

除了打开和关闭文件外，头文件<stdio.h>还提供了写文件和读文件的函数。常见的写文件的函数包括 fputc（写入一个字符）、fputs（写入一个字符串）、fprintf（类似于函数 printf，以格式化的方式写入整数、浮点数、字符串等）和 fwrite（以字节的方式写入无格式的数据）等。各函数的说明如下。

(1) int fputc(int c,FILE * stream)；

fputc 函数有两个参数，分别表示要写入的字符 c 和要写入数据的文件指针 stream。该函数将字符 c 写入文件指针 stream 指向的文件，并将写入的字符作为结果返回，如果写入过程中出现错误，该函数返回文件结束标志 EOF。

(2) int fputs(const char * string,FILE * stream)；

fputs 函数的两个参数分别是要写入的字符串 string 和要写入数据的文件指针 stream。该函数将字符串 string 写入文件指针 stream 指向的文件，如果写入成功，函数返回非负的整数，否则返回文件结束标志 EOF。

(3) int fprintf(FILE * stream,const char * format [,argument]…)；

fprintf 函数与 printf 函数相似，只是 fprintf 函数多了一个参数 stream 以表示要写入数据的文件。该函数的功能是将数据按照格式控制串 format 写入文件指针 stream 指向的文件中，如果写入成功，函数返回写入的字节数，否则返回一个负数表示错误。

(4) size_t fwrite(const void * buffer,size_t size,size_t count,FILE * stream)；

fwrite 函数可以把从内存中指定位置开始的指定个数的字节以二进制的方式写入文件。函数 fwrite 有 4 个参数。其中，第一个参数 buffer 是一个指针，指向内存中要写入文件的数据的首地址；第二个参数 size 是要写入文件的数据对象的大小，类型是 size_t(size_t 是一个 C 标准的类型，其具体的实现在不同的语言版本中不同，在大多数情况下实现为无符号整型)，一般使用运算符 sizeof 计算数据对象所占空间的字节数；第三个参数提供要写入的数据对象的个数；第四个参数是文件指针，它指明了要写入数据的文件。函数 fwrite 可以一次将从 buffer 开始的，size * count 个字节的数据写入指针 stream 指向的文件中。如果写入成功，该函数返回写入的数据对象的个数，如果发生了错误，函数的返回值肯定小于

count。

例16-2将音频媒体的信息写入一个二进制文件中。该程序首先以"wb"的方式(因为程序要以二进制的方式将音频媒体的信息写入文件)建立或打开了一个名为am.dat的文件,如果文件打开成功,则往文件中写入各音频媒体对象,最后关闭文件。

【例16-2】 编写程序将音频媒体对象写入文件中。

```
//文件 ex16_2.cpp:将对象写入文件
#include <stdio.h>
#include <string.h>

#define MAX_STR_LEN 32

class AudioMedia
{
public:
    AudioMedia(int num,char * n,char * s)
    {
        no=num;

        strncpy(name,n,MAX_STR_LEN - 1);

        name[MAX_STR_LEN - 1]=0;

        strncpy(singer,s,MAX_STR_LEN - 1);
        singer[MAX_STR_LEN - 1]=0;
    }

    ~ AudioMedia() { }

private:
    int no;                                 //序号
    char name[MAX_STR_LEN];                 //媒体名
    char singer[MAX_STR_LEN];               //歌手
};

int main()
{
    //声明媒体对象
    AudioMedia am1(1,"The Colour of My Love","Celine Dion");
    AudioMedia am2(2,"Thriller","Michael Jackson");

    //声明文件指针
    FILE * fptr;

    //打开文件
    if ((fptr=fopen("am.dat","wb"))!=NULL)
    {
        //以二进制方式写入数据
        fwrite(&am1,sizeof(AudioMedia),1,fptr);
        fwrite(&am2,sizeof(AudioMedia),1,fptr);
        //关闭文件
```

```
            fclose(fptr);
        }
        else
            printf("Open file or create file error.\n");

        return 0;
    }
```

程序运行后，在程序运行的当前目录下有一个文件 am.dat，如果以文本编辑器打开该文件，发现其中有一些和媒体对象相关的文本字符，还有一些非文本字符，所有这些数据就是以二进制方式写入的对象。

例 16-2 中的程序在打开文件后写入对象 am1 和 am2 时使用的是 fwrite 函数：

```
fwrite(&am1,sizeof(AudioMedia),1,fptr);
fwrite(&am2,sizeof(AudioMedia),1,fptr);
```

函数 fwrite 的第一个参数是一个指针，指向内存中要写入文件的数据的首地址。这里要写入的是对象 am1 和 am2，表达式 &am1 和 &am2 就是获取对象 am1 和 am2 在内存中的首地址。第二个参数是要写入文件的对象的大小，使用运算符 sizeof 计算类 AudioMedia 的对象所占空间的字节数。函数 fwrite 也可以一次将一个数组中的多个元素写入文件，写入一个数组时，需要给函数 fwrite 的第一个参数提供数组的首地址，第三个参数则提供数组元素的个数。因为在这里程序一次只需要写入一个对象，相当于写入一个数组元素个数为 1 的数组，所以第三个参数为 1。函数 fwrite 的第四个参数是文件指针，指明要写入数据的文件。

类 AudioMedia 的一个对象包括一个整型的数据成员 no 和两个字符数组 name、singer，函数 fwrite 可以一次将一个完整的对象写入文件，实际上就是将一个整数和两个字符数组写入文件，不管类 AudioMedia 的对象中存储的整数 no 有多大，两个字符串分别是什么，它在内存中都只占固定大小，即 sizeof(AudioMedia)个字节，函数 fwrite 也按照该对象在内存中“原始数据”的格式(二进制格式)将该固定大小的连续存储字节写入文件中。如果使用格式化方式将一个无符号整数写入文件，则该整数可能会占一个字节(如该数小于 10)、两个字节(如该数大于等于 10 且小于 100)或多个字节(大于等于 100)。由于函数 fwrite 写入文件的数据所占的字节数总是可以计算的(一个整数总是占 sizeof(int)个字节)，很多时候会发现要重新读出用 fwrite 写入文件的数据要比读出用 fprintf 写入文件的数据要方便得多。

和例 15-22 相比，例 16-2 将类 AudioMedia 的数据成员 name 和 singer 都定义为字符数组，而不是字符指针。请考虑，如果将 name 和 singer 依然定义为字符指针，例 16-2 中的程序还能将类 AudioMedia 的对象一次性写入文件吗？如果不行，该怎么修改？

16.2.3 读文件

类似于写文件的函数，头文件<stdio.h>也提供了一些从文件中读取数据的函数，包括 fgetc(从文件中读取一个字符)、fgets(读取一个字符串)、fscanf(类似于函数 scanf，以格

式化的方式从文件中读取整数、浮点数、字符串等)和 fread(以字节的方式读取数据)等。为了方便文件内容的读取,头文件<stdio.h>还提供了在读文件时判断是否到达文件尾的函数 feof。各函数的说明如下。

(1) int fgetc(FILE * stream);

与 fputc 相反,fgetc 函数从文件指针 stream 指向的文件的当前位置读取一个字符,以 int 类型返回,如果出错或已经到达了文件结束的位置,将返回文件结束标志 EOF。

(2) char * fgets(char * string,int n,FILE * stream);

与 fputs 相反,fgets 函数从文件指针 stream 指向的文件的当前位置开始读取字符串,判断字符串结束位置的条件是:遇到换行符(读入该换行符)、到达文件结束位置或者读取了 n−1 个字符。读取的字符串存入 string 所指的内存单元中,并在所有读取的字符之后添加字符串结束标记'\0'。如果读取成功,函数返回 string,如果出错或读取前已经到达了文件结束的位置,将返回 NULL。

(3) int fscanf(FILE * stream,const char * format [,argument]...);

与 fprintf 相反,fscanf 函数与 scanf 函数也非常相似,只是 fscanf 函数多了一个参数 stream 以指向要读取数据的文件。该函数的功能是从指定文件中将数据按照格式控制串 format 读出并转换成相应的类型以存入对应的参数中,如果读取成功,该函数返回转换成功的参数的个数,如果出错或读取前已经到达了文件结束的位置,将返回文件结束标志 EOF。

(4) size_t fread(void * buffer,size_t size,size_t count,FILE * stream);

与 fwrite 相反,函数 fread 的作用是从文件的当前位置读取指定字节数的数据放入内存的指定位置。第一个参数 buffer 是一个指针,指向内存中要写入数据的位置;函数 fread 的后 3 个参数与函数 fwrite 的 3 个参数的意义相同,给出了要读取数据对象的字节数、数目及被读取的文件。函数 fread 可以从指定文件的当前位置一次性读取 size * count 个字节的数据并存入 buffer 中。如果读取成功,该函数返回成功读取的数据对象的数目;如果发生错误,函数的返回值小于 count。

(5) int feof(FILE * stream);

函数 feof 判断 stream 指向的文件是否已经到达了文件的结束位置,如果是,返回非 0 值;否则,返回 0。

下面例子将例 16-1 写入文件 test.txt 中的字符串分别按行和按单词读出并输出到屏幕上。

【例 16-3】 编写程序把例 16-1 中程序创建的文件 test.txt 中的数据读取出来。

```
//文件 ex16_3.cpp: 打开文件并读取其中的数据
#include <stdio.h>

#define MAX_BUF_LEN 80

int main()
{
    //定义 FILE 指针
```

```
    FILE * fptr;
    char buf[MAX_BUF_LEN];
    //打开文件
    if ((fptr=fopen("test.txt","r"))!=NULL)
    {
        //从文件中读取一个字符串
        fgets(buf,MAX_BUF_LEN,fptr);
        printf("Data from File: %s\n",buf);
        //关闭文件
        fclose(fptr);
    }
    else
        printf("Open file or create file error.\n");
    //再次打开文件
    if ((fptr=fopen("test.txt","r"))!=NULL)
    {
        printf("Read words one by one:\n");
        while (!feof(fptr))
        {
            fscanf(fptr,"%s",buf);
            printf("\t%s\n",buf);
        }
        //关闭文件
        fclose(fptr);
    }
    else
        printf("Open file or create file error.\n");
    return 0;
}
```

运行结果：

```
Data from File: This is a test file.
Read words one by one:
        This
        is
        a
        test
        file.
```

例16-3中的程序声明文件指针fptr后，调用fopen函数打开文件test.txt。文件打开方式为"r"，因为打开的文件是文本文件，而且打开文件的目的是为了读取其中的数据。文件test.txt由例16-1中的程序创建，如果例16-1和例16-3中的程序不在同一个目录下，则

需要将文件 test.txt 复制到例 16-3 中程序所在目录。程序第一次打开文件后,调用 fgets 函数读取文件中的数据,语句

```
fgets(buf,MAX_BUF_LEN,fptr);
```

从文件的当前位置读取 MAX_BUF_LEN－1(79)个字符,或者读取数据直到遇到换行符或文件尾,读取的数据放到字符数组 s 中。这里,文件 test.txt 中的所有数据被一次读出。程序在输出读出的数据后关闭文件。

接下来,程序再次调用 fopen 函数打开文件 test.txt,并利用循环逐个读取其中的单词。语句

```
while (!feof(fptr))
```

调用函数 feof 判断进行文件读操作时是否到达文件尾,到达文件尾则跳出循环。语句

```
fscanf(fptr,"%s",buf);
```

从 fptr 指向的文件中读取一个字符串存入 buf,碰到空白符或换行符时结束。文件 test.txt 中各单词以空格符分开,该语句一次读出一个单词。实际执行时,程序在最后一次循环读入并输出“file.”后,将回到循环测试条件处,此时执行 feof(fptr)函数调用就发现到达了文件末尾,返回非 0 值,结束整个 while 循环。

例 16-3 中的程序从文本文件中读取数据,下面例子中的程序则重新打开了例 16-2 中程序创建并写入数据的文件 am.dat,并且按照写入数据的格式将文件中的数据读取出来。

【例 16-4】 编写程序将例 16-2 中程序创建的文件 am.dat 中的数据读取出来。

```
//文件 ex16_4.cpp: 读二进制文件
#include <stdio.h>
#include <string.h>
#define MAX_STR_LEN 32
class AudioMedia
{
public:
    AudioMedia() {};
    AudioMedia(int num,char * n,char * s)
    {
        no=num;
        strncpy(name,n,MAX_STR_LEN-1);
        name[MAX_STR_LEN-1]=0;
        strncpy(singer,s,MAX_STR_LEN-1);
        singer[MAX_STR_LEN-1]=0;
    }
    ~AudioMedia(){}
    void showInfo()
    {
```

```
        printf("No.%d: %s,%s\n",no,name,singer);
    }
private:
    int no;                                  //序号
    char name[MAX_STR_LEN];                  //媒体名
    char singer[MAX_STR_LEN];                //歌手
};
int main()
{
    AudioMedia am1,am2;
    //声明文件指针
    FILE * fptr;
    //打开文件
    if ((fptr=fopen("am.dat ","rb"))!=NULL)
    {
        //读取对象
        fread(&am1,sizeof(AudioMedia),1,fptr);
        fread(&am2,sizeof(AudioMedia),1,fptr);
        am1.showInfo();
        am2.showInfo();
        //关闭文件
        fclose(fptr);
    }
    else
        printf("File open error.\n");
    return 0;
}
```

运行结果：

```
No.1: The Colour of My Love,Celine Dion
No.2: Thriller,Michael Jackson
```

例16-4中的程序仍然需要定义类AudioMedia，因为要从文件中读取类AudioMedia的对象，为了输出读取出来的对象的信息，程序为类AudioMedia增加了成员函数showInfo。主程序首先调用fopen函数以"rb"方式（文件中包含的是二进制数据，需要以二进制方式打开读取）打开文件am.dat，文件am.dat由例16-2中的程序创建，如果例16-2和例16-4中的程序不在同一个目录下，则需要将文件am.dat复制到例16-4中程序所在目录。

打开文件成功后，程序调用fread函数从文件中读取类AudioMedia的两个对象，并分别放入am1和am2中：

```
fread(&am1,sizeof(AudioMedia),1,fptr);
fread(&am2,sizeof(AudioMedia),1,fptr);
```

函数fread的第一个参数指向内存中要写入数据的位置，这里需要把数据读入对象am1和am2中，所以第一个参数应该分别是am1和am2的首地址。函数fread的后3个参数与函数fwrite的后面3个参数相同，分别给出要写入或读出数据的字节数以及指向被操作文件的指针。在例16-4中，函数fread一次性读出固定大小个字节(类AudioMedia的一个对象所占的空间大小)，这些字节的“原始数据”被放到变量am1和am2所在的内存空间，而这些字节也正好是例16-2中用函数fwrite写入的一个完整的对象，这种方法可以确保读出的对象和写入时完全一致。

文件操作完毕之后，需要调用fclose函数关闭文件。

上面两个例子在操作文件时都是打开文件后从头到尾将一个文件的数据全部读完，在从文件读取数据时，程序怎么知道应该从文件的什么位置读取数据？是否只能按照从头到尾的先后顺序读取文件呢？答案在于文件打开后的文件位置指针。使用fopen函数打开文件后，返回的FILE结构维护了一个指向文件当前操作位置的指针——文件位置指针(如图16-2所示)。进行文件的读写操作时，程序从文件位置指针所指向的位置读取或写入数据。每进行一次文件的读写操作，文件的位置指针会自动根据读写操作所涉及的字节数进行调整，如执行语句

```
c=fgetc(fptr);
```

后，由于函数fgetc的功能是从文件中读取一个字符，占一个字节的空间，指针fptr所指向的FILE结构中的文件位置指针会自动增1，指向下一个将被读取的字节。

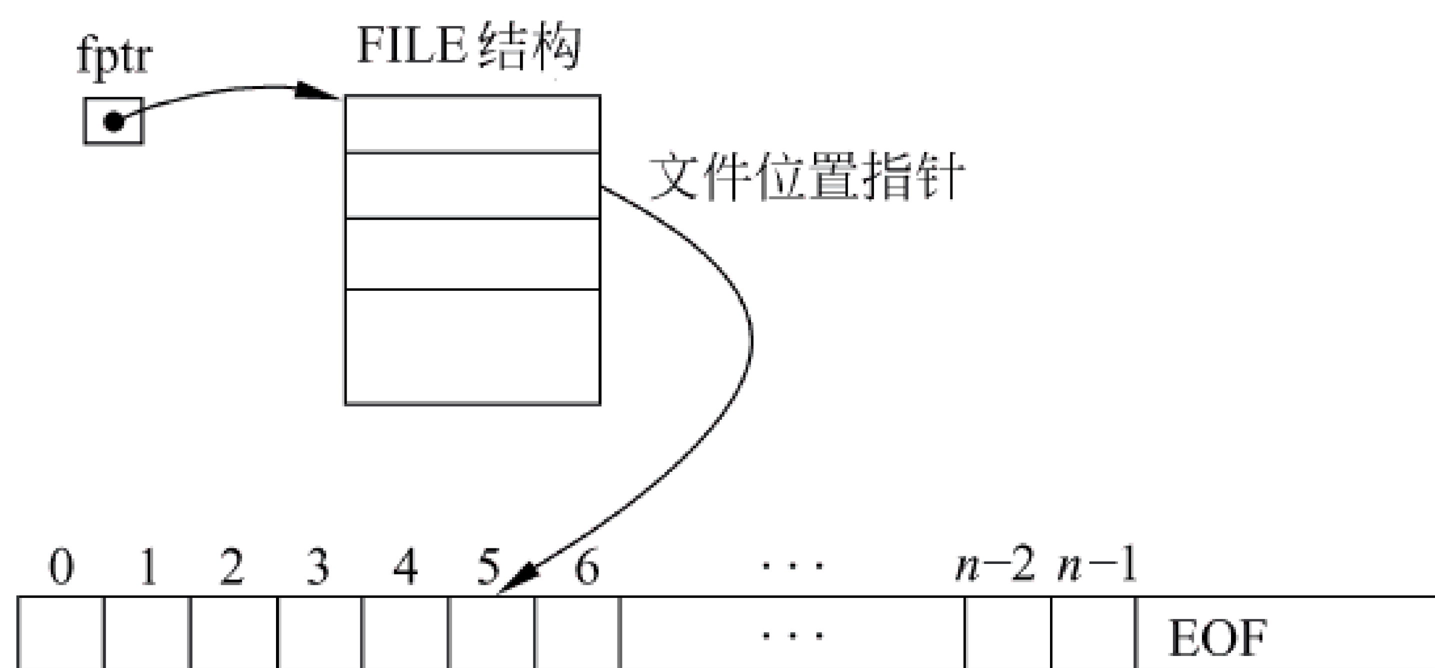


图16-2　文件位置指针指向文件当前操作位置

如果不能改变文件位置指针的值，文件的读写只能按顺序进行。如果需要从文件中随机读取一些数据该怎么办呢？头文件<stdio.h>中的fseek函数可以解决这个问题。

函数fseek的定义为：

```
int fseek(FILE *,long,int);
```

其中，第一个参数为文件指针；第二个参数是一个长整型，表示从第三个参数所确定的位置开始向前或向后(依赖于第二个参数的正负值)的字节数；第三个参数是定位文件位置指针的基准，其值可以是SEEK_SET、SEEK_CUR和SEEK_END 3个值之一，分别表示从文件的起始位置、文件位置指针的当前位置和文件的尾部开始。例16-5中的程序读取文件am.dat中的数据，但跳过了第一个对象，直接读取第二个对象的内容。

【例16-5】 编写程序将例16-2创建的文件am.dat中的所有数据读取出来，要求跳过第一个对象，直接读取第二个对象。

```
//文件 ex16_5.cpp: 读二进制文件
#include <stdio.h>
#include <string.h>
#define MAX_STR_LEN 32
class AudioMedia
{
public:
    AudioMedia(){};
    AudioMedia(int num,char *n,char *s)
    {
        no=num;
        strncpy(name,n,MAX_STR_LEN-1);
        name[MAX_STR_LEN-1]=0;
        strncpy(singer,s,MAX_STR_LEN-1);
        singer[MAX_STR_LEN-1]=0;
    }
    ~AudioMedia(){}
    void showInfo()
    {
        printf("No.%d: %s,%s\n",no,name,singer);
    }
private:
    int no;                                    //序号
    char name[MAX_STR_LEN];                    //媒体名
    char singer[MAX_STR_LEN];                  //歌手
};
int main()
{
    AudioMedia audio;
    //声明文件指针
    FILE *fptr;
    //打开文件
    if ((fptr=fopen("am.dat","rb"))!=NULL)
    {
        //修改文件位置指针,跳过一个对象
        fseek(fptr,sizeof(AudioMedia),SEEK_SET);
        //读取对象
        fread(&audio,sizeof(AudioMedia),1,fptr);
        audio.showInfo();
        //关闭文件
        fclose(fptr);
    }
```

```
        else
            printf("File open error.\n");
        return 0;
}
```

运行结果：

```
No.2: Thriller,Michael Jackson
```

例16-5中的程序在打开文件am.dat成功后，执行语句：

```
fseek(fptr,sizeof(AudioMedia),SEEK_SET);
```

该语句修改文件位置指针的值，使之从文件的开始位置(第三个参数SEEK_SET)向后跳过一个对象(sizeof(AudioMedia)个字节)，指向存储第二个对象的位置，保证后续语句都能读到正确的值。

16.3 通过文件流进行文件操作

C++除了保留ANSI C通过FILE结构进行文件操作的机制外，还提供了使用对象进行文件操作的功能。一个C++程序运行时，和输入输出相关的cout、cin、cerr和clog 4个对象会被自动创建，与这些对象相关联的流为程序提供了与特定的文件或设备进行通信的通道。例如，cout对象(标准输出流对象)使程序向显示器输出数据，cin对象(标准输入流对象)使程序从键盘读取数据，cerr和clog对象(标准错误流对象)使程序向显示器输出错误信息。

除了这些特定功能的预定义对象外，C++还允许程序员为特定的文件创建输入输出流对象来完成对该文件的各种操作。

16.3.1 打开和建立文件

要在C++中进行文件处理，需要包含头文件<fstream.h>。头文件<fstream.h>中包含了3个文件流类的定义。其中，类ifstream实现文件的输入，类ofstream实现文件的输出，类fstream实现文件的输入输出。只需要声明这些类的对象就可以进行文件的各种操作。这3个类是分别从类istream、ostream和iostream中派生而来，因此可以像使用cout、cin这些对象一样来操作文件。

输入输出类的继承关系如图16-3所示。

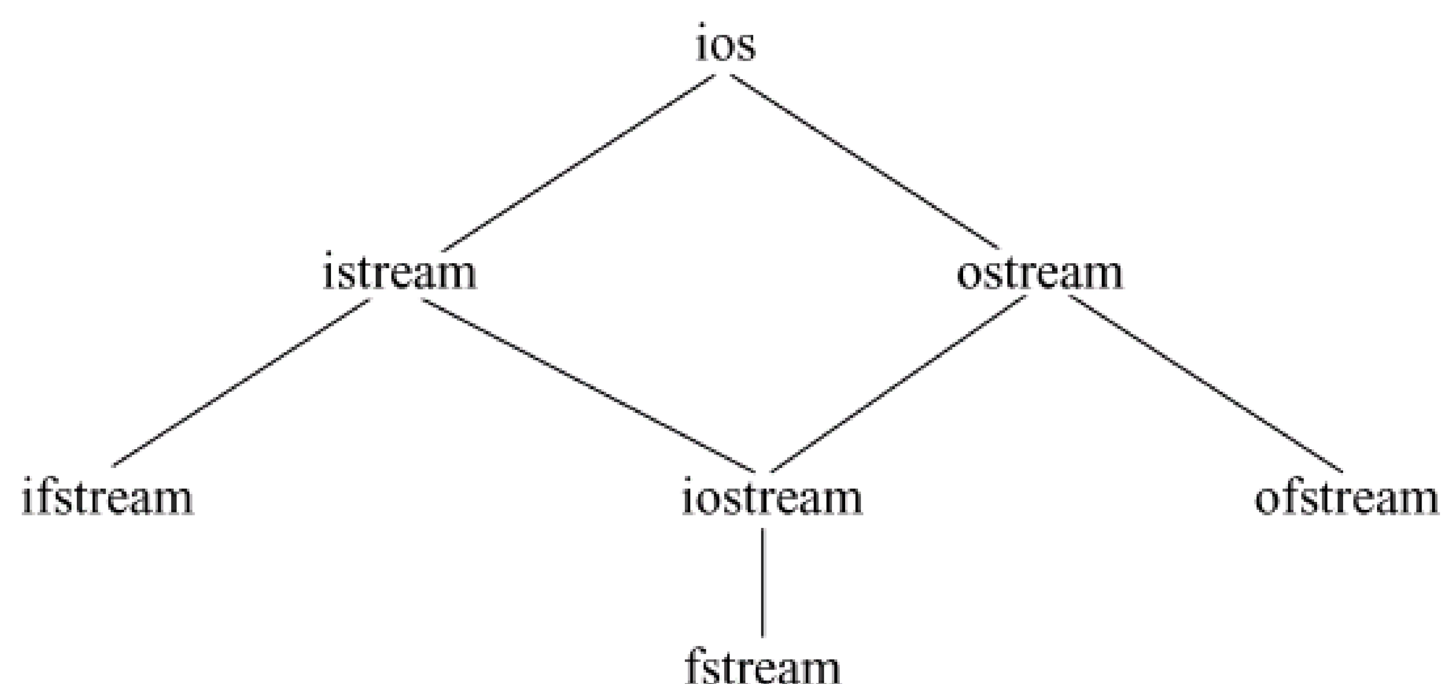


图16-3 输入输出流的继承关系

在 C++ 中打开和建立文件是非常方便的。声明一个文件流对象,并让该流对象和某个文件关联起来就可以打开或创建一个文件。

【例 16-6】 编写程序使用文件流新建或打开一个文件,并写入字符串"This is a test file."。

```
//文件 ex16_6.cpp: 新建和打开文件
#include <fstream.h>

int main()
{
    //声明文件流对象
    ofstream outFile("test.txt",ios::out);

    if (!outFile)
    //使用错误流对象输出错误信息
        cerr<<"Open file or create file error."<<endl;
    else
    //输出数据到与对象 outFile 关联的文件中
        outFile<<"This is a test file.";

    return 0;
}
```

程序运行后,在程序运行的当前目录多了一个文件 test.txt。打开该文件,发现其中的数据为"This is a test file."。

例 16-6 中 main 函数的第一条语句:

```
ofstream outFile("test.txt",ios::out);
```

声明了类 ofstream 的一个对象 outFile,并提供了初始化参数"test.txt"和 ios::out。其中,第一个参数是要打开或创建的文件名;第二个参数则是文件打开方式。参数 ios::out 表示以写的方式打开文件,并废弃文件当前内容,如果指定文件不存在,则创建该文件。创建对象 outFile 时,类 ofstream 的构造函数将对象 outFile 和文件 test.txt 关联起来(以 ios::out 的方式打开该文件)。表 16-2 列出了类 fstream 所有文件打开方式。

表 16-2 类 fstream 打开文件的方式

打开方式	描述
ios::in	打开一个供读取的文件
ios::out	打开一个供写入的文件
ios::app	写入的所有数据将被追加到文件的末尾,此方式需要使用 ios::out
ios::ate	写入的数据将被追加到文件的末尾,但也可写到其他地方,此方式不需使用 ios::out
ios::trunc	废弃当前文件内容
ios::nocreate	如果要打开的文件并不存在,那么以此参数调用 open()函数将无法进行
ios::noreplace	如果要打开的文件已存在,试图用 open()函数打开时将返回一个错误
ios::binary	以二进制的形式打开一个文件

类 ofstream 的对象的默认功能就是输出数据到文件,创建类 ofstream 的对象时即使不提供第二个参数 ios::out 也表示打开用于输出的文件。所以,语句

```
ofstream outFile("test.txt");
```

也表示打开或创建文件 test.txt 用于输出。声明类 ofstream 的对象时也可以暂时不关联(打开)特定的文件,然后在需要的时候通过 open 成员函数来打开特定的文件并建立与该文件的关联关系。所以,上述语句也可以用下面两条语句替代:

```
ofstream outFile;
outFile.open("test.txt",ios::out);
```

例 16-6 中的程序在创建 ofstream 对象 outFile 并执行打开文件操作后,对文件打开成功与否进行了测试。语句

```
if (!outFile)
    cerr<<"Open file or create file error."<<endl;
```

调用了定义于类 ios 中的运算符成员函数 operator!来判断文件打开操作是否成功。如果文件打开不成功,则输出错误信息“Open file or create file error.”;否则,执行语句:

```
outFile<<"This is a test file.";
```

该语句调用流插入运算符重载函数 operator <<将字符串"This is a test file."写入文件中。

和使用 FILE 结构操作文件相比,例 16-6 中的程序打开文件后没有显式地关闭文件。程序执行完毕后,main 函数终止,对象 outFile 被撤销,在执行其析构函数时会关闭与其相关联的文件 test.txt。程序也可以使用成员函数 close 显式地关闭 ofstream 对象关联的文件,相应的语句为:

```
outFile.close();
```

一般情况下,程序应该尽早显式地关闭不再使用的文件,这可以减少程序执行时所占用的资源,也会使程序的结构更加清晰。

例 16-6 中的程序将文件 test.txt 按文本文件操作。和使用 FILE 结构操作文件一样,使用文件流的方式也可以二进制的方式操作文件,后面将予以讨论。

16.3.2 写文件

C++ 向文件写入格式化数据和在屏幕上输出格式化数据的方法相同,都可以借助定义于类 ostream 中的<<运算符重载函数来完成。不同之处在于,对于前者类 ostream 的对象关联的是磁盘文件,而对于后者则关联的是显示器。下面例子使用文件流的方法将不同数据类型的数据写到文件中。

【例 16-7】 编写程序使用文件流的方式将整数、浮点数、字符串等类型的数据写入文件中。

```
//文件 ex16_7.cpp:采用文件流的方式写文件
#include <fstream.h>
```

```
int main()
{
    //创建文件流对象
    ofstream outFile("test.txt",ios::out);
    //判断对象 outFile 打开文件成功与否
    if (!outFile)
    //使用错误流对象输出错误信息
        cerr<< "Open file or create file error."<< endl;
    else
    {
        //调用流插入运算符函数写文件
        outFile<< 5<< "string"<< " "<< 1.2;
        //显式关闭文件
        outFile.close();
    }
    return 0;
}
```

程序运行后，在程序运行的当前目录下有一个文件 test. txt，打开该文件，可以看到文件中的数据为“5string 1. 2”。

例 16-7 中的 main 函数在声明 ofstream 的对象 outFile 的同时，执行了打开文件 test. txt 的操作。如果打开文件成功，就可以像使用对象 cout 一样来使用对象 outFile，只是对象 cout 将格式化数据输出到屏幕上，而对象 outFile 则将格式化数据写入和它相关联的文件 test. txt 中。语句

```
outFile<< 5<< "string"<< " "<< 1.2;
```

先将整数 5(以字符'5'的形式)写入文件，接着写入字符串"string"，为了将字符串"string"与后续写入的浮点数 1. 2(将以字符串"1. 2"的形式写入)区分开来，程序在它们之间写入了一个空格。写文件的操作完成之后，程序立即调用 close 成员函数关闭打开的文件 test. txt，断开和该文件的关联关系。运行程序后，用文本编辑器打开文件 test. txt，不难发现，其中的数据内容和格式("5string 1. 2")与语句

```
cout<< 5<< "string"<< " "<< 1.2;
```

在屏幕上输出的数据和格式完全相同。

16.3.3 读文件

C++ 可以使用类 ofstream 的对象和定义于类 ostream 中的<<运算符重载函数进行写文件的操作，也可以使用类 ifstream 的对象和定义于类 istream 中的>>运算符重载函数从文件中读取数据。下面例子中的程序以读的方式打开了例 16-7 中的程序建立的文件 test. txt，并按照正确的格式读出其中的数据。

【例 16-8】 编写程序使用文件流的方式将例 16-7 中程序创建的文件 test. txt 中的所有数据读取出来。

```
//文件 ex16_8.cpp: 以文件流的方式读文件
#include <fstream.h>
#define MAX_STR_LEN 32

int main()
{
    //创建文件流对象
    ifstream inFile("test.txt",ios::in);

    int a;
    char b[MAX_STR_LEN];
    float c;

    //判断对象 inFile 打开文件成功与否
    if (!inFile)
    //使用错误流对象输出错误信息
        cerr<< "File open error."<< endl;
    else
    {
        //调用流提取运算符重载函数从文件中读取数据
        inFile>> a>> b>> c;

        //输出读取的数据
        cout<< "INTEGER - - "<< a<< endl;
        cout<< "STRING - - "<< b<< endl;
        cout<<      "FLOAT - - "<< c<< endl;

        //显式关闭文件
        inFile.close();
    }

    return 0;
}
```

运行结果：

```
INTEGER - - 5
STRING - - string
FLOAT - - 1.2
```

例 16-8 的 main 函数首先声明了一个类 ifstream 的对象 inFile，并同时打开文件 test.txt。声明类 ifstream 的对象时，可以同时打开一个指定的文件，也可以在声明对象时不打开任何文件，而在后面通过调用成员函数 open 建立和指定文件之间的关联关系。即语句

```
ifstream inFile("test.txt",ios::in);
```

可以被替换成下面两条语句：

```
ifstream inFile;
inFile.open("test.txt",ios::in);
```

由于类 ifstream 的对象的默认操作是打开文件用于输入，因此参数 ios::in 可以省略。

语句

```
ifstream inFile ("test.txt");
```

同样能起到打开文件 test. txt 用于输入的作用。如果文件打开成功，则程序按照例 16-7 中程序写入数据的格式将数据逐个读出。语句序列

```
inFile>> a>> b>> c;
cout<< "INTEGER - - "<< a<< endl;
cout<< "STRING - - "<< b<< endl;
cout<< "FLOAT - - "<< c<< endl;
```

从文件中读取一个整数、一个字符串和一个浮点数，并将它们输出到屏幕上。程序在完成对文件的所有操作后调用成员函数 close 显式地关闭打开的文件。

除了通过流插入运算符和流提取运算符进行文件的输入输出外，类 istream 和 ostream 都提供了重定位文件位置指针的函数，类 istream 的成员函数 seekg 以及类 ostream 的成员函数 seekp 的用法和头文件＜stdio. h＞中的函数 fseek 的用法相似，定义于类 istream 中的成员函数 seekg 的函数原型为：

```
istream & seekg(long,ios::seek_dir);
```

其第二个参数表示定位文件位置指针的基准，其值可以为 ios：：beg（默认值，表示相对于文件的开始位置定位）、ios：：cur（相对于文件位置指针的当前位置定位）和 ios：：end（相对于文件尾定位）。第一个参数是一个长整型，表示偏移量。

下面例子中的程序打开文件 test. txt 后，没有按顺序读取文件中的所有内容。在读取整数 a 后，程序使用类 istream 中的成员函数 seekg 重定位文件位置指针，跳过了后面的字符串"string"，然后直接读取后面的浮点数 1. 2。

【例 16-9】 编写程序使用文件流的方式读取例 16-7 中程序创建的文件 test. txt 中的数据，跳过其中的字符串"string"。

```
//文件 ex16_9.cpp: 读文件
# include < fstream.h>

int main()
{
    //创建文件流对象
    ifstream inFile("test.txt",ios::in);

    int a;
    float c;

    if (!inFile)
        cerr<< "File open error."<< endl;
    else
    {
        //读取一个整数
        inFile>> a;

        //修改文件位置指针，往后跳 7个字节
```

```
            inFile.seekg(7,ios::cur);
            //读取一个浮点数
            inFile>>c;
            //输出读取的数据
            cout<<"INTEGER--"<<a<<endl;
            cout<<"FLOAT--"<<c<<endl;
            //关闭文件
            inFile.close();
        }
        return 0;
}
```

运行结果:

```
INTEGER--5
FLOAT--1.2
```

例16-9中的程序在读取文件中的数据时为了跳过字符串"string"修改了文件位置指针。语句:

```
inFile.seekg(7,ios::cur);
```

将和流对象inFile相关联的文件的文件位置指针设置为当前位置之后的7个字节,跳过了字符串"string"和一个空格符。

C++的文件操作除了可以使用<<和>>对文件中有格式的数据进行方便地读写外,还可以利用定义于类ostream中的write成员函数和类istream中的read成员函数,以二进制的方式读写数据(操作二进制文件),甚至可以用文件保存自定义类的对象(对象也是自定义类型的变量)。

下面例子中的程序使用文件流的方式实现了对类AudioMedia的对象的保存和恢复。

【例16-10】 编写程序实现使用文件流的方式保存和恢复自定义类的对象。

```
//文件ex16_10.cpp:使用文件流以二进制方式读写对象
#include<fstream.h>
#include<string.h>
#define MAX_STR_LEN 32
class AudioMedia
{
public:
    AudioMedia(){};
    AudioMedia(int num,char * n,char * s)
    {
        no=num;
        strncpy(name,n,MAX_STR_LEN-1);
        name[MAX_STR_LEN-1]=0;
```

```
        strncpy(singer,s,MAX_STR_LEN-1);
        singer[MAX_STR_LEN-1]=0;
    }
    ~AudioMedia(){}
    void showInfo()
    {
        cout<<"No."<<no<<": "<<name<<","<<singer<<endl;
    }
    void setName(char * n)
    {
        strncpy(name,n,MAX_STR_LEN-1);
        name[MAX_STR_LEN-1]=0;
    }
private:
    int no;                                         //序号
    char name[MAX_STR_LEN];                         //媒体名
    char singer[MAX_STR_LEN];                       //歌手
};
int main()
{
    //声明类 AudioMedia 的对象 audio
    AudioMedia audio(1,"The Colour of My Love","Celine Dion");
    //输出对象 audio
    cout<<"Object audio:"<<endl;
    audio.showInfo();
    cout<<endl;
    //创建文件流对象,同时创建用于保存对象的文件
    ofstream outFile("am.dat",ios::out|ios::binary);
    if (!outFile)
        cerr<<"Open file or create file error."<<endl;
    else
    {
        //将对象保存到文件中
        outFile.write((char*) &audio,sizeof(audio));
        //关闭文件
        outFile.close();
    }
    //修改 audio 的 name
    audio.setName("Miracle");
    //输出修改后的对象 audio
    cout<<"Modified object audio:"<<endl;
    audio.showInfo();
```

```
        cout<<endl;

        //创建文件流对象,再次打开该文件用于读
        ifstream infile("am.dat",ios::in|ios::binary);
        if (!outFile)
            cerr<<"File open error."<<endl;
        else
        {
            //从文件中恢复对象 audio
            infile.read((char * ) &audio,sizeof(audio));
            infile.close();
        }

        //输出重新读取的对象 audio
        cout<<"After read object audio from file:"<<endl;
        audio.showInfo();
        cout<<endl;

        return 0;
}
```

运行结果:

```
Object audio:
No.1: The Colour of My Love,Celine Dion

Modified object audio:
No.1: Miracle,Celine Dion

After read object audio from file:
No.1: The Colour of My Love,Celine Dion
```

例 16-10 中的 main 函数首先声明了类 AudioMedia 的对象 audio,将其数据成员分别初始化为 1、"The Colour of My Love"和"Celine Dion",并调用成员函数 showInfo 输出该对象的信息。接下来,语句

```
ofstream outFile ("am.dat",ios::out | ios::binary);
```

声明了文件流类 ofstream 的对象 outFile,并同时打开二进制文件 am.dat,打开方式为 ios::out|ios::binary。参数 ios::out|ios::binary 表示打开或创建一个供写入的二进制文件,如果文件存在则废弃文件当前内容。指明文件打开方式的参数可以是用或运算符|连接起来的不冲突的多种打开方式,如打开一个既可以读也可以写的文件,打开方式可以是 ios::in|ios::out。

然后,程序对打开文件的结果进行了测试。如果文件打开成功,则使用语句

```
outFile.write((char * )&audio,sizeof(audio));
```

将对象 audio 写入文件,并显式地关闭文件。定义于类 ostream 的 write 成员函数的定义为:

```
ostream& write(char * pch,int nCount);
```

它带两个参数，要求第一个参数为一个字符指针，指向要写入文件的数据的首地址，第二个参数是一个整数，给出了要写入文件的字节数。程序需要将对象 audio 写入文件，所以第一个参数应该是对象 audio 的首地址，但需要进行类型的强制转换，第二个参数则是对象 audio 所占的空间大小（字节数）。

关闭文件后，程序调用类 AudioMedia 的成员函数 setName 将对象 audio 的数据成员 name 修改为" Miracle "，并再次输出对象 audio 的信息，显示其新设置的值。

前面的程序将类的对象保存到文件中，后面的程序则将从文件中恢复该对象。语句

```
ifstream infile("am.dat",ios::in | ios::binary);
```

声明了类 ifstream 的对象 infile，并再次以读的方式打开文件 am. dat，打开文件后程序调用成员函数 read 将保存在文件中的对象读出到对象 audio 中。定义于类 istream 的 read 成员函数的定义为：

```
istream& read(char * pch,int nCount);
```

它也有两个参数，第一个参数是一个字符指针，指向要存放读入数据的存储单元的首地址；第二个参数是一个整数，给出了要读取的数据的字节数。

程序在将对象写入文件和从文件中读出对象时，都按照“原始数据”（二进制）的格式进行，这样可以保证对象数据的正确性。

习　题　16

16.1　填空：

(1) 文件流类的________成员函数的功能是关闭文件。

(2) 文件流类的________成员函数的功能是打开一个文件。

(3) 类 istream 和类 ostream 中的成员函数________和________的功能是重定位文件位置指针。

(4) 头文件<iostream. h>中声明的标准流对象包括________、________、________和________。

(5) 使用 FILE 结构操作文件时，一般使用函数________和________读写二进制数据。

16.2　编写程序建立文件 mytest. txt，然后从键盘读取字符写入该文件，直到读入文件结束标志。

16.3　假设有一个文件 one. dat 已经存在，编写程序创建文件 two. txt，并将文件 one. dat 中的内容复制到文件 two. txt 中。

16.4　一个食品店包含一系列的商品，如表 16-3 所示。

表 16-3　习题 16.4 的商品信息表

序　号	名　称	数　量	价　格
1	瓜子	20	2.50
2	蛋糕	8	6.00
3	饼干	15	4.50
4	牛奶	30	1.80

为了方便查询商店现有商品的数量和价格，编写一个程序将所有商品及相关信息保存到文件 sell. dat 中，并实现如下功能：

(1) 销售一些商品后修改文件中相应商品的数量。

(2) 采购新商品后保存新商品的信息到文件中。

(3) 进货后修改文件中相应商品的数量。

(4) 查询商店所有商品的信息。

16.5　下面是一个三角形类 Triangle 的定义:

```
class Triangle {
public:
    Triangle(double a,double b,double c) {
        x=a;
        y=b;
        z=c;
    }
    int saveToFile(ostream &outFile);
    int loadFromFile(istream * inFile);
private:
    double x,y,z;
};
```

(1) 实现成员函数 saveToFile,将对象自身保存到文件中。

(2) 实现成员函数 loadFromFile,从文件中恢复对象。

(3) 编写程序使用类 Triangle,并实现如下功能:

① 将类 Triangle 的多个对象保存到一个指定文件中。

② 能一次从指定文件中恢复多个对象。

16.6　编写一个学生类 Student,要求包含一个学生所有的基本信息和基本操作,并尝试编写程序使用类 Student 管理自己所在班级的所有同学,学生的信息以对象的形式保存在文件中。

16.7　编写一个用于英语单词学习的系统,按照如下步骤进行:

(1) 编写类 Eword,类 Eword 的对象表示一个英文单词及其相关的所有信息。

(2) 为类 Eword 添加显示对象信息的成员函数。

(3) 为类 Eword 添加保存对象到文件以及从文件恢复对象的成员函数。

(4) 编写类 WordList,以链表或数组的方式管理类 Eword 的多个对象。

(5) 为类 WordList 添加查询一个单词的成员函数。

(6) 为类 WordList 添加插入一个单词(类 Eword 的对象)的成员函数。

(7) 为类 WordList 添加逐个显示英语单词的成员函数。

(8) 编写程序利用上述两个类借助文件管理自己要学习的英语单词。

(9) 对程序进行进一步的优化,如为已经记住的单词添加标记,下次不再显示等,使系统更加实用。

第17章 异 常

【学习内容】

本章介绍异常处理。主要内容包括：

- ◆ 异常处理的优点。
- ◆ 异常的抛出和传播。
- ◆ 异常的捕获和处理。

【学习目标】

- ◆ 理解异常处理的意义。
- ◆ 理解 C++ 的异常处理机制。
- ◆ 掌握异常的传播机制。
- ◆ 掌握异常的捕获规则。
- ◆ 掌握使用异常处理机制处理错误的方法。
- ◆ 知道何时需要使用异常处理机制来处理错误。

C++ 的可扩充性给程序设计带来了很多优势，但同时也可能大大增加发生错误的次数和种类。本章将介绍异常处理机制，以编写更清晰、更健全、更具容错性的程序。

17.1 异常处理的意义

不同的软件产品对错误处理的要求不同，为自己编制的软件一般会忽略很多程序异常，而作为商业化的产品则必须提供更多的错误处理代码。错误处理代码是程序的重要组成部分，并占有相当的比例。处理错误的方法很多。例如，前面章节中的一些程序，用运算符 new 动态分配了空间之后，一般都要对指针值进行测试以判断空间分配是否成功，打开文件后也要判断文件打开是否成功，如果操作不成功则要进行错误处理。一般情况下，错误处理代码分布在整个系统的不同地方。

采用这种错误处理方法的好处是可以直接看到错误处理的情况，但同时使程序本身的代码（完成主要功能的代码）和错误处理代码混在一起，就有可能使本来就难懂的程序变得更加难懂，对程序代码的理解和维护也变得更加困难。

在程序的编写过程中，应该为每一种可能的错误提供错误处理方法，并且在错误产生时要提供清楚有意义的错误信息。C++ 的异常处理机制可以做到这一点。程序运行时常见

的异常,包括使用 new 运算符时无法获得所需要的内存空间、指针,或者数组下标越界、算术运算时产生溢出、除数为 0 等。

C++ 的异常处理机制可以将程序的“主要代码”和错误处理代码分离开来,从而提高程序的可读性和可维护性。通过编写异常处理程序,系统可以捕获所有类型的异常,或者根据需要只捕获特定类型或某些类型的异常。捕获异常之后,异常处理程序可以对错误进行适当的处理,而不是任其发生并造成后果。如果产生致命的错误,而且程序没有提供适当的处理措施,程序就会终止运行。因此,使用异常处理机制减少程序中未能捕获的错误,可以使程序更加健壮。

异常处理可以使系统从导致异常的错误中恢复。错误产生时程序抛出异常,异常处理模块则捕获异常并进行适当的处理。异常处理常常无法使程序恢复到正常执行的轨道上来,但它能在程序无法恢复时为程序提供有序的整理操作,从而使程序可以正常结束。

异常处理提高了程序的容错能力。编写程序时,在可能出错的地方只需要抛出异常即可,从而把精力集中到程序的主要流程之中,之后再提供相对独立的错误处理代码捕获并处理异常。

17.2 异常处理基础

C++ 的异常处理一般用于可能产生错误的函数(或模块)不去处理错误或无法处理错误的情况,这些函数通常只抛出异常而不去处理它。如果有异常处理模块能捕捉到该异常则处理它,如果没有则程序终止。

C++ 的异常处理涉及 3 个关键字: try、throw 和 catch。编程时,将可能出错并产生异常的代码放在 try 块中,try 块的后面则跟着一个或多个 catch 块(异常处理器),每个 catch 块捕捉和处理一种异常。如果 try 块有异常抛出,则程序控制离开 try 块,在其后的 catch 块中逐个搜索合适的异常处理器(异常处理器的搜索顺序稍后介绍)。如果 try 块没有异常抛出,则跳过 catch 块,执行最后一个 catch 块之后的语句。

函数中可以包含抛出异常的 throw 语句,也可以不抛出任何异常。程序的 try 块可以直接抛出异常(使用 throw 语句)或者通过其直接或间接调用的函数抛出异常,抛出异常的 throw 语句称为抛出点。抛出异常之后,程序控制无法再返回到抛出点。

下面是异常处理的一个简单的例子。

【例 17-1】 编写一个除法函数,要求在除数不为零时返回正确的结果,除数为零时抛出异常,并编写程序使用不同的参数调用该函数,捕捉使用不恰当的参数时函数抛出来的异常。

```
//ex17_1.cpp: 除数为零的异常例子
#include <iostream.h>
#include <string.h>
//定义异常类 MyException,当除数为 0时抛出该异常类的对象
class MyException
{
public:
```

```
    MyException(char * str)
    {
        msg=new char[strlen(str)+1];
        strcpy(msg,str);
    }
    MyException(const MyException &e)
    {
        msg=new char[strlen(e.msg)+1];
        strcpy(msg,e.msg);
    }
    ~MyException()
    {
        if (msg!=NULL)
            delete msg;
    }
    void show()
    {
        cout<<msg<<endl;
    }
private:
    char * msg;
};
//定义除法函数 division,除数为 0 时抛出异常
double division(int dividend,int divisor)
{
    if (divisor==0)
    //抛出异常对象
        throw MyException("error: divided by zero!");
    return (double)dividend/divisor;
}
//测试程序
int main()
{
    int a,b;
    double result;
    cout<<"Enter two integers (end-of-file to end): ";
    while (cin>>a>>b)
    {
        //在 try 中调用函数 division,如果抛出异常将被 catch 捕获
        try
        {
            result=division(a,b);
            //如果有异常抛出,则跳出 try 块
```

```
            cout<<a<<"/"<<b<<"="<<result<<endl;
        }
        //捕捉 MyException类型的异常
        catch (MyException &e)
        {
            cout<<"Caught MyException: ";
            e.show();
        }
        cout<<"Enter two integers (end-of-file to end):";
    }

    return 0;
}
```

运行结果：

```
Enter two integers (end-of-file to end): 12 7
12/7=1.71429
Enter two integers (end-of-file to end): 2 0
Caught MyException: error: divided by zero!
Enter two integers (end-of-file to end): 34 5
34/5=6.8
```

例17-1定义了异常类MyException作为程序抛出异常的类型。类MyException的构造函数需要一个字符指针参数，可以把程序出错时的信息放到异常对象中然后抛出。throw可以抛出任意类型的异常，如抛出一个整型类型的异常(整数)或抛出一个浮点类型的异常(浮点数)等。类MyException还定义了一个复制构造函数。

程序的第一次输出显示函数division执行成功，没有抛出异常。第二次输入时除数为0，程序在调用函数division时执行语句：

```
throw MyException("error: divided by zero!");
```

该语句以字符串"error: divided by zero!"为参数创建了类MyException的一个临时对象，并抛出该对象。注意，这里try块并没有显式地包含抛出异常的任何语句(throw语句)，而是在调用函数division时，在函数division中包含了抛出异常的语句。一般来说，抛出异常的语句可以直接在try块中体现，也可以通过try块中的函数调用甚至是深层嵌套的函数调用来体现。

程序的try块的后面跟了一个catch块，语句

```
catch (MyException &e)
```

表示该catch块捕捉MyException类型的异常，这正好和throw语句所抛出的异常类型相同。catch块中的e是前面throw语句产生的对象的引用。该catch块捕捉到异常后，输出异常对象中携带的信息后(通过类MyException的公有成员函数show)结束。一个catch块可以很复杂，完成很多的整理工作，也可以很简单，什么事都不做。

17.3　异常的抛出和传播

关键字 throw 通常带一个操作数，表示要抛出的异常。throw 可以带任何类型的操作数，包括自定义类型。如果被抛出的是一个对象，则一般也称为异常对象。异常抛出后，指定捕获相应类型的最近的一个异常处理器捕获该异常，try 块的异常处理器（catch 块）紧接在 try 块的后面。如果抛出异常后，找不到类型符合的异常处理器，系统就会调用 terminate 函数，terminate 函数默认调用 abort 函数终止程序的执行。

抛出异常时，throw 语句生成异常对象的一个副本，即一个临时对象，异常处理器执行完毕后删除该临时对象。

【例 17-2】　编写一个程序，抛出整型、浮点类型、双精度类型的异常，然后使用异常处理器捕捉它们。

```
//ex17_2.cpp: 抛出多种类型异常的例子
#include <iostream.h>

int main()
{
    int a,myInt;
    float myFloat;
    double myDouble;

    cout<<"Enter a integer (end-of-file to end): ";
    while (cin>>a)
    {
        //根据读取的整数的值分别抛出不同类型的异常
        try
        {
            switch (a%3)
            {
                case 0:
                    //输入整数为 3的倍数时抛出整型异常
                    myInt=a;
                    throw myInt;
                    break;
                case 1:
                    //输入整数为 3的倍数加 1时抛出 float类型异常
                    myFloat=(float)a;
                    throw myFloat;
                    break;
                case 2:
                    //输入整数为 3的倍数加 2时抛出 double类型异常
                    myDouble=a;
                    throw myDouble;
                    break;
```

```
                default:
                    break;
            }
        }
        //捕获整型异常
        catch (int e)
        {
            cout<<"Integer Exception: "<<e<<endl;
        }
        //捕获浮点类型异常
        catch (float e)
        {
            cout<<"Float Exception: "<<e<<endl;
        }
        //捕获双精度类型异常
        catch (double e)
        {
            cout<<"Double Exception: "<<e<<endl;
        }
        cout<<"Enter a integer (end-of-file to end): ";
    }
    return 0;
}
```

运行结果:

```
Enter a integer (end-of-file to end): 10
Float Exception: 10
Enter a integer (end-of-file to end): 11
Double Exception: 11
Enter a integer (end-of-file to end): 12
Integer Exception: 12
Enter a integer (end-of-file to end): 13
Float Exception: 13
```

例 17-2 中的程序根据读入的整数的大小分别抛出整型异常、浮点类型异常和双精度类型的异常。

程序运行时第一次输入的整数为 10,try 块抛出了一个浮点类型的异常,异常抛出后,程序控制在 try 块后面逐个寻找合适的异常处理器,第一个异常处理器能捕捉的异常类型是 int 类型,和抛出异常的类型不符合。接下来的异常处理器是 float 类型,和抛出异常类型一致,因此抛出的异常被第二个异常处理器捕捉到,该异常处理器输出了异常信息后结束。如果一个 try 块后面有多个异常处理器,只需将它们连续按顺序排列在后面就可以了,中间不需要加分号,每个 catch 块也不需要像 switch 结构那样用 break 语句跳出控制快,抛出的异常被捕捉后,程序执行完相应的异常处理器后即跳过所有的 catch 块,执行最后一个 catch 块的后续语句。

程序的第二次输入的整数为11,try块抛出一个double类型的异常,被第三个异常处理器捕捉到,并输出了该异常的信息。

第三次的输入为12,程序抛出的整型类型的异常被第一个异常处理器捕捉到。

异常只能在try块中抛出,并由紧跟在try块后面的符合类型的catch块捕捉。如果在try块外面抛出异常,该异常将不会被捕捉到,系统就会调用terminate函数终止程序的运行。异常处理器可以什么都不干,也可以做一些清理工作。产生异常之后可以不结束程序的运行,但必须跳出抛出异常的程序块,并且程序控制无法再返回到抛出点。异常可以在try块中显式抛出,也可以是在try块中调用的函数或多层嵌套调用的函数中抛出。

try块可以嵌套,对于内层try块中抛出的异常,程序控制首先在内层try块后面的catch块中寻找符合的异常处理器,如果找到则进行处理,异常不再往外传播。如果内层try块中抛出的异常在内层try块后面的catch块中找不到合适的异常处理器,程序控制将该异常向外传播,到外层try块后面的catch块中寻找合适的异常处理器,找到后进行处理。如果异常传播到最外层的try块仍然找不到合适的异常处理器,则程序调用terminate函数。

下面例子中的程序演示了异常在嵌套异常处理模块中的传播。

【例17-3】 编写一个处理两个整数相加的函数,如果结果在0～128之间则返回正确值,如果结果过大则抛出整型类型异常,如果结果过小则抛出字符串类型的异常。整型异常在本函数中处理,字符串类型的异常传播到外层处理,并编写程序测试该函数。

```
//ex17_3.cpp:嵌套异常处理例子
#include <iostream.h>
//add函数,结果过大或过小时都抛出异常
int add(int a,int b)
{
    int res;
    try
    {
        res=a+b;
        if (res>128)
        //抛出整型异常
            throw res;
        if (res<0)
        //抛出字符串异常
            throw "Negative result!";
    }
    //捕捉整型异常
    catch (int e)
    {
        cout<<"The result is too large: "<<e<<endl;
        return -1;
    }
    return res;
}
```

```
int main()
{
    int a,b,result;
    cout<<"Enter two integers (end-of-file to end): ";
    while (cin>>a>>b)
    {
        try
        {
            result=add(a,b);
            if (result>=0)
                cout<<a<<"+"<<b<<"="<<result<<endl;
        }
        //捕捉传播到外层的所有异常
        catch (...)
        {
            cout<<"Unexpected exception found."<<endl;
        }
        cout<<"Enter a integer (end-of-file to end): ";
    }
    return 0;
}
```

运行结果：

```
Enter two integers (end-of-file to end): 12 66
12+66=78
Enter a integer (end-of-file to end): 76 89
The result is too large: 165
Enter a integer (end-of-file to end):-145 36
Unexpected exception found.
```

例17-3中的函数add根据两个整数相加的结果分别抛出两种异常，当结果太大时抛出整型异常，太小时抛出字符串异常。但是，add函数中的异常处理器却只能捕捉一种类型的异常——整型异常，如果try块抛出整型异常，try块后面的异常处理器可以捕获并处理，如果try块抛出字符串异常，则不会被其后的异常处理器捕获，该异常就会向外传播。

由于函数add可能向外传播异常，主函数main就需要将调用该函数的语句放在一个try块中，并在该try块后面使用异常处理器捕捉传播出来的所有异常，这里使用的语句是：

```
catch (…)
```

其中，catch的参数“…”表示该异常处理器捕捉所有类型的异常。

程序运行时第一次输入的两个整数之和为78，在0～128之间，add函数返回了正确的结果。第二次输入的两个整数之和大于128，函数add里面的try块抛出的整型异常被其后的异常处理器捕捉到。第三次输入的两个整数之和小于0，函数add的try块抛出字符串异常，由于其后的异常处理器不能捕捉字符串异常，该异常向外传播，到main函数中的try块，被main函数中try块后面的异常处理器捕捉到并处理。

17.4　异常的捕获和处理

异常处理器在 catch 块中，catch 块以关键字 catch 开始，接下来的括号内是参数类型和参数名。参数类型给出了异常处理器能捕捉到的异常的类型，参数名可选。如果指定了参数名，则可以在异常处理器内引用这个参数。catch 块传递捕捉的异常对象的方式和函数调用是传递参数的方式类似。如果异常是一个对象而不是简单类型的值，要注意捕获异常时，传递参数的过程。以例 17-1 为例，其 catch 块是：

```
catch (MyException &e) {异常处理代码}
```

采取的是传引用的方式，因此 catch 块中的参数 e 就是前面 throw 语句产生的对象的引用。如果采取对象传值方式，即

```
catch (MyException e) {异常处理代码}
```

则捕获异常时，就将调用类 MyException 的复制构造函数（例 17-1 确实也为类 MyException 定义了一个复制构造函数），用前面 throw 语句产生的对象再生成一个新的对象 e。这时，参数 e 是抛出的异常对象的一个副本，如果抛出的异常对象是一个局部动态对象，则程序控制转到异常处理器中时，异常对象已被撤销。

下面的异常处理器

```
catch (int e){异常处理代码}
```

可以捕捉整型异常。catch 块也可以不指定参数名(括号中只有参数类型，无参数名)，则异常对象不从抛出点传递到异常处理器中，异常处理器也就不能获取异常信息。例如：

```
catch (int){ 异常处理代码 }
```

这个异常处理器仍能捕获 int 类型的异常，但不能获得抛出的整型异常对象的值的大小。

程序按顺序搜索寻找匹配的异常处理器，抛出的异常将被 try 块后面第一个类型符合的异常处理器捕获。如果 try 块后面没有能捕获该异常的异常处理器，该异常将会向外层传播，到外面一层的 try 块后面接着寻找匹配的异常处理器，这个过程一直持续直到跳出所有 try 块或找到匹配的异常处理器。没有被捕获的异常将调用 terminate 函数，terminate 函数默认调用 abort 函数终止程序的执行，程序员也可以使用 set_terminate 函数指定 terminate 函数将调用的函数，从而使得存在不被捕获的异常时能调用自己指定的函数。

catch 后面的括号中也可以没有参数类型和参数名，而只有一个省略号。即

```
catch (…) {异常处理代码}
```

这样定义的异常处理器可以捕捉任何类型的任何异常。

一个异常被抛出后，也许同时会有几个异常处理器和该异常匹配，这时程序执行第一个匹配的异常处理器。如果有多个异常处理器都能匹配某一种类型的异常，则异常处理器的排列顺序会影响到异常处理的结果。

异常处理器的参数类型和抛出的异常类型之间满足下面条件之一时，异常被捕捉。

(1) 异常处理器的参数类型和抛出异常的类型完全相同。

(2) 异常处理器的参数类型是抛出的异常对象的基类。

(3) 异常处理器的参数是基类的指针或引用,抛出异常的类型是派生类的指针或引用。

(4) 异常处理器的参数是 void * 类型的指针,抛出异常的类型是某一种类型的指针。

(5) catch 处理器为 catch(…)。

异常捕获需要准确的类型匹配,进行异常的类型匹配时只允许派生类向基类转换,而不允许其他的类型转换或提升,如 int 类型不能转换为 float 类型(float 类型的异常处理器不能捕获 int 类型的异常)。

例 17-4 定义了具有继承关系的两个类,后面的例子将抛出和捕获这两个自定义类型的异常对象。

【例 17-4】 具有继承关系的两个类。

```
//文件 ex17_4.h: 定义继承关系的异常类
#include <iostream.h>

//定义基类
class Base
{
public:
    //定义函数输出基类信息
    void show()
    {
        cout<< "Base object."<< endl;
    }
};

//定义派生类
class Derived: public Base
{
public:
    //重定义 show 函数输出派生类信息
    void show()
    {
        cout<< "Derived object."<< endl;
    }
};
```

上面程序定义了具有继承关系的两个简单的类,基类 Base 只有一个公有成员函数 show,用来显示其对象的信息,派生类 Derived 则重定义了基类的 show 函数,显示派生类对象自身的信息。下面例子中的程序分别抛出类 Base 和类 Derived 的异常对象,并尝试捕获它们。

【例 17-5】 编写程序抛出例 17-4 中定义的类的异常对象,并使用异常处理器捕获它们。

```
//ex17_5.cpp: 抛出基类和派生类异常
#include "ex17_4.h"
```

```
int main()
{
    int no;
    cout<<"Please input an integer: ";
    while (cin>>no)
    {
        try
        {
            if ((no%2)==0)
            //抛出基类对象
                throw Base();
            else
            //抛出派生类对象
                throw Derived();
        }
        catch (Base &b)
        {
            cout<<"Exception: ";
            b.show();
        }
        catch (Derived &d)
        {
            cout<<"Exception: ";
            d.show();
        }
        cout<<"Please input an integer: ";
    }
    return 0;
}
```

运行结果：

```
Please input an integer: 1
Exception: Base object.
Please input an integer: 2
Exception: Base object.
Please input an integer: 3
Exception: Base object.
Please input an integer: 4
Exception: Base object.
Please input an integer:
```

例17-5中的程序从键盘读取整数，当读入的整数为奇数时抛出派生类(Derived)类型的异常。语句

```
throw Derived();
```

调用类 Derived 的构造函数创建一个临时对象并抛出。当读入的整数为偶数时抛出基类(Base)类型的异常,语句

```
throw Base();
```

创建基类 Base 的一个临时对象并抛出。程序中 try 块后面跟着两个异常处理器,捕获基类异常的异常处理器放在捕获派生类异常的异常处理器前面。

程序运行时的 4 次输入导致程序两次抛出基类异常,两次抛出派生类异常,但由于基类异常和派生类异常都可以与参数是基类类型的异常处理器匹配,因此所有的异常都被第一个异常处理器捕获。

要想使每个异常处理器都能准确地捕获该类型的异常,捕获派生类异常的异常处理器必须放在捕获基类异常的异常处理器前面。例 17-5 中的程序如果希望派生类异常被参数为派生类类型的异常处理器捕获,只需将两个异常处理器调换位置即可。

习　题　17

17.1　异常处理机制一般用于什么时候?在编程的过程中,何时使用一般的错误处理方法,何时使用异常处理机制?

17.2　如果异常抛出后在 try 块后面没有找到与该异常匹配的异常处理器,会发生什么情况?

17.3　抛出异常后是否一定会终止程序的运行?什么时候需要终止程序的运行?

17.4　编写一个程序,使用 new 运算符动态分配空间,当空间分配失败时抛出异常,并捕获处理该异常,处理异常时要考虑不能有内存泄漏。针对本题讨论使用异常处理机制和使用一般错误处理方法各自的优势。

17.5　创建对象时有时在构造函数中需要动态分配空间,如 String 类,而分配动态空间的操作可能会失败,这会导致创建的对象不可用,但人们又无法通过构造函数的执行结果获得创建对象的结果(构造函数没有返回值),试编写程序使用异常处理机制弥补这一缺陷。

17.6　编写一个 C++ 程序,显示抛出各种指针类型的异常,并使用异常处理器捕获它们,考虑参数为 void * 的异常处理器。

17.7　编写程序使用 set_terminate 函数将自定义函数设置成系统调用 terminate 时执行的函数,并编程测试之。

17.8　编写一个 C++,从嵌套函数调用的各个层次中抛出异常,并在不同的层次捕获它们,体验异常的传播机制。

17.9　编写一个 C++ 程序,用异常处理机制处理数组下标越界的错误。

17.10　异常处理器中也可以再抛出异常,如果一个异常处理器抛出和该异常处理器同类型的异常,是否会造成无穷递归?试编写程序验证之。

第 18 章 模 板

【学习内容】

本章介绍 C++ 的模板机制。主要内容包括：

- ◆ 类属机制。
- ◆ 函数模板的定义与使用。
- ◆ 类模板的定义与使用。

【学习目标】

- ◆ 了解类属程序设计的概念。
- ◆ 理解模板和模板的实例化机制。
- ◆ 掌握模板的定义和使用方法。

模板是 C++ 实现代码重用的一种有效的机制。模板通过类型参数化，即把类型定义为参数，从而提高了代码的可重用性。C++ 提供了两种模板：一种是函数模板；另外一种是类模板。本章将介绍类属机制的概念，并介绍 C++ 支持类属编程的函数模板和类模板的定义与使用方法。

18.1 类 属 机 制

回顾第 6 章的例 6-4，为了实现对 int 数组和 double 数组的排序，重载了两个同名函数，这些函数的原型是：

```
void sortArray(int b[],int len);
void sortArray(double b[],int len);
void displayArray(int b[],int len);
void displayArray(double b[],int len);
```

即对 sortArray 和 displayArray 进行了重载。再仔细研究两个 sortArray 函数的内部实现，就会发现它们的执行流程几乎完全一样，唯一不同的是被排序的数据的类型不一样。这两个重载的函数既然如此相似，能否将二者抽象成一个函数，而将处理的数据的类型作为参数来处理呢？对于 displayArray 函数，也会有同样的思考。

同样，在第 11 章中实现了 Stack 类，该类实现了元素是 int 类型的栈。如果需要一个元素是 double 类型的栈，Stack 就不好用了。这时需要重新设计一个类来实现元素是 double

类型的栈。可以想象,针对 double 类型设计的新类与 Stack 类也是非常相似的,差别也仅在于元素的类型不同。那么,是否同样能够将二者抽象成一个类,而将元素的类型作为参数来处理呢?

在 C++ 中,确实可以实现上述的想法。对于某些程序,从它们的逻辑功能看,彼此非常相似,所不同的主要是处理对象(数据)的类型。如果可以提供具有相同逻辑功能的程序代码(将相似性抽象出来),然后将数据类型作为参数,这样不仅能够大幅度地节约代码,而且也极大地方便了程序的维护。这就是类属编程(generic programming)(又称泛型编程)的思想。类属编程在 C++ 中是通过模板(template)机制来实现的。模板分为函数模板和类模板两种,以所处理的数据类型的说明作为参数的类称为类模板,而以所处理的数据类型的说明作为参数的函数称为函数模板。

通过使用模板可以使程序具有更好的代码重用性。模板是对源代码进行重用,而不是通过继承和复合重用对象代码。当用户使用模板时,参数由编译器来替换。模板参数可以由类型参数或非类型参数组成,类型参数可用 class 和 typename 关键字来指明,二者的意义相同,都表示后面的参数名代表一个潜在的标准类型或用户定义的类型,非类型参数由一个普通参数声明构成。

下面分别介绍函数模板和类模板的定义和使用。

18.2 函数模板

从例 6-4 可以看出,重载函数可以实现对不同类型的数据施加相似的操作。但是,重载函数的代码是非常相似的,这种冗余一方面增加了编程的负担,另一方面也为程序的维护带来了隐患。通过模板机制,可以将这些重载的函数抽象成一个函数模板(function template),该模板实际上就是一个函数的代码,只不过函数中某些数据的类型没有明确给出,而是作为参数来处理。使用函数模板时,用具体的数据类型替换模板参数,就可以得到处理相应类型数据的函数,这一过程也称为对函数模板进行实例化(template instantiation),这样生成的函数称为函数模板实例(function template specialization)。函数模板本身并不能够被执行,只有通过实例化得到的函数模板实例才是程序执行的实体,在使用时函数模板实例与一般的函数完全一样。函数模板与函数模板实例的关系如图 18-1 所示。

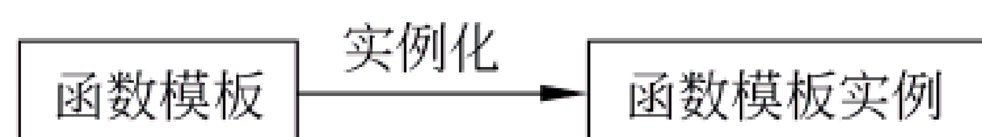


图 18-1　函数模板与函数模板实例的关系

18.2.1 函数模板的定义

在 C++ 程序中,函数模板的定义都是以关键字 template 开头,该关键字之后使用尖括号<>括起来的是模板的形式参数列表,每个形式参数之前都有一个关键字 class 或 typename,形式参数名为有效的标识符。例如:

```
template<class T>
template<class ElementType>
template<class T1,class T2>
```

模板的形式参数名可以用来说明函数的参数的类型、函数的返回值类型和函数内部的局部变量的类型。模板的每个形式参数要在函数的参数列表中至少出现一次。形式参数名的作用域局限于函数模板的范围内,不同的函数模板可以定义相同的形式参数名。

程序设计风格提示:模板参数的名字一般都是大写字母开头,如果名字由多个单词组成,每个单词的首字母大写,其他字母小写。

模板的参数定义之后是函数模板的定义,函数模板的定义与一般函数的定义类似,只不过在定义函数模板时,可以使用模板的形式参数来说明函数的参数、返回值和局部变量。

下面以例 6-4 的问题为例,用函数模板来定义不同类型的数组的排序和输出。

```
template<class ElementType>
void sortArray(ElementType b[], int len)
{
    for (int i=0; i<len-1; i++)    {
        //在未排序序列中找到最小元素
        int min=i;
        for (int j=i+1; j<len; j++)
            if (b[j]<b[min])
                min=j;
        //将找到的最小元素与第 i个元素交换,即存放到排序序列末尾
        ElementType temp=b[i];
        b[i]=b[min];
        b[min]=temp;
    }
}
template<class ElementType>
void displayArray(ElementType b[], int len)
{
    for ( int index=0; index<=len-1; index++)
        if (index != len-1)
            cout<<b[ index ]<<"\t";
        else
            cout<<b[ index ]<<endl;
}
```

这两个函数模板将数组元素的类型抽象成模板的参数 ElementType,函数模板定义中用 ElementType 来声明函数的形式参数的数组元素类型。实际使用函数模板时,模板的形式参数将被实际的具体数据类型所替换,从而将函数模板实例化成具体的函数,真正被调用执行的是这些实例化的函数,即函数模板实例。

下面将介绍函数模板是如何被实例化和使用的。

18.2.2　使用函数模板

一般来说,函数模板定义时,明确规定了对数据的处理流程,而某些数据类型则作为模板的参数需要等到模板实例化时再确定具体的类型。而函数模板的实例化由编译器来完成。当编译器处理一个函数调用时,如果该函数是一个函数模板,那么编译器根据函数调用

的实在参数的类型确定每个模板形式参数的对应的具体数据类型,通过将函数模板中的模板形式参数替换成相应的数据类型,编译器可以定义函数模板的实例,而该函数调用所调用的就是实例化得到的模板函数。从这个过程也可以看出,只有模板的每个形式参数在函数模板的参数列表中至少出现一次,才可能根据函数调用语句的实在参数确定所有的模板形式参数对应的数据类型,从而实现函数模板的实例化。

下面通过一个示例程序来说明函数模板的使用。在这个程序中,将使用上一节定义的两个函数模板来实现例6-4的功能。

【例18-1】 用函数模板实现元素为不同类型(double和int)的数组的排序和输出。

```
//ex18_1: 用函数模板实现不同类型 (double和 int)的数组的排序和输出
#include< iostream.h>
#define SIZE 8
template< class ElementType>
void sortArray(ElementType b[], int len)
{
    for (int i=0; i<len-1; i++)    {
        //在未排序序列中找到最小元素
        int min=i;
        for (int j=i+1; j<len; j++)
            if (b[j]<b[min])
                min=j;
        //将找到的最小元素与第 i个元素交换,即存放到排序序列末尾
        ElementType temp=b[i];
        b[i]=b[min];
        b[min]=temp;
    }
}
template< class ElementType>
void displayArray(ElementType b[], int len)
{
    for ( int index=0; index<=len-1; index++ )
        if ( index !=len-1 )
            cout<<b[ index ]<<"\t";
        else
            cout<<b[ index ]<<endl;
}
int main()
{
    int ai[ SIZE ]={ 18, 35, 36, 61, 9, 112, 77, 12};
    double af[ SIZE ]={ 12.1,-23.8, 3.7,-16.0, 9.1, 12.12, 7.7, 56.3};
    //用 int实例化 displayArray函数模板
    //调用该实例函数输出 ai排序前的数据
    cout<<"Before sorting:\n";
    cout<<"ai: \t";
```

```
    displayArray(ai, SIZE);
    //用 int 实例化 sortArray 函数模板
    //调用该实例函数对 ai 进行排序
    sortArray(ai, SIZE);
    //用 int 实例化 displayArray 函数模板
    //调用该实例函数输出 ai 排序后的结果
    cout<<"After sorting:\n";
    cout<<"ai: \t";
    displayArray(ai, SIZE);
    //用 double 实例化 displayArray 函数模板
    //调用该实例函数输出 af 排序前的数据
    cout<<"Before sorting:\n";
    cout<<"af: \t";
    displayArray(af, SIZE);
    //用 double 实例化 sortArray 函数模板
    //调用该实例函数对 af 进行排序
    sortArray(af, SIZE);
    //用 double 实例化 displayArray 函数模板
    //调用该实例函数输出 af 排序后的结果
    cout<<"After sorting:\n";
    cout<<"af: \t";
    displayArray(af, SIZE);
    return 0;
}
```

程序执行结果：

```
Before sorting:
ai:  18     35     36     61     9     112     77     12
After sorting:
ai:  9      12     18     35     36    61      77     112
Before sorting:
af:  12.1   -23.8  3.7    -16    9.1   12.12   7.7    56.3
After sorting:
af:  -23.8  -16    3.7    7.7    9.1   12.1    12.12  56.3
```

显然，程序的执行效果与例 6-4 完全一样，而本例程序更加简洁，更容易维护。因为，如果需要修改排序算法，在例 6-4 中必须修改两个非常类似 sortArray 重载函数，而在本例中，只要修改函数模板 sortArray 即可。

例 18-1 说明了函数模板实例化及调用的过程。

当 C++ 编译器处理到 main 函数的语句

```
displayArray(ai,SIZE);
```

时，因为实在参数 ai 是 int 数组，而函数模板 displayArray 的第一个形式参数是 ElementType b[]，其中 ElementType 是模板参数，所以就用 int 替换整个 displayArray 函

数模板中的所有 ElementType,将该函数模板实例化成一个输出 int 数组的所有元素的完整函数,即函数模板实例:

```
void displayArray(int b[],int len)
{
    for (int index=0;index<=len-1;index++)
        if (index!=len-1)
            cout<<b[index]<<"\t";
        else
            cout<<b[index]<<endl;
}
```

该函数模板实例将被编译,调用语句实际调用的就是上面建立的实例函数。在排序后,输出排序结果的函数调用语句

```
displayArray(ai,SIZE);
```

也被处理成对这个函数模板实例的调用。

当编译器处理到语句

```
sortArray(ai,SIZE);
```

时,根据同样的理由,编译器用 int 替换整个 sortArray 函数模板中的所有 ElementType,将该函数模板实例化成一个将 int 数组的元素进行从小到大排序的完整函数。

```
void sortArray(int b[], int len)
{
    for (int i=0; i<len-1; i++)
    {
        //在未排序序列中找到最小元素
        int min=i;
        for (int j=i+1; j<len; j++)
            if (b[j]<b[min])
                min=j;

        //将找到的最小元素与第 i 个元素交换,即存放到排序序列末尾
        int temp=b[i];
        b[i]=b[min];
        b[min]=temp;
    }
}
```

原来的调用语句实际调用的就是这个实例函数。类似地,对于语句

```
displayArray(af,SIZE);
```

编译器用数组 af 的元素的类型 double 来替换整个 displayArray 函数模板中的所有 ElementType,并将该函数模板实例化成一个输出 double 数组的所有元素的函数模板实例。

```
void displayArray(double b[],int len)
```

```
{
    for (int index=0;index<=len-1;index++)
        if (index!=len-1)
            cout<<b[index]<<"\t";
        else
            cout<<b[index]<<endl;
}
```

而语句

```
sortArray(af,SIZE);
```

将导致编译器用 double 替换整个 sortArray 函数模板中的所有 ElementType，将该函数模板实例化成一个将 double 数组的元素进行从小到大排序的函数模板实例。

```
void sortArray(double b[], int len)
{
    for (int i=0; i<len -1; i++)
    {
        //在未排序序列中找到最小元素
        int min=i;
        for (int j=i+1; j<len; j++)
            if (b[j]<b[min])
                min=j;
        //将找到的最小元素与第 i个元素交换,即存放到排序序列末尾
        double temp=b[i];
        b[i]=b[min];
        b[min]=temp;
    }
}
```

从上面的分析可以看出，函数模板与重载函数有非常密切的联系。一个函数模板可能被实例化成多个函数模板实例，这些函数模板实例重载的函数，而编译器也正是采取重载的方法来确定每个函数调用到底是调用哪个实例，即根据实在参数的类型、数目及顺序来调用相应的函数。

模板提供了很好的软件重用的支持，也有利于软件的一致性维护。但也要注意到，模板虽然只编写一次，但是编译后的代码实际上仍然包含了多个实例化函数的代码。

18.3　类　模　板

C++ 除了支持函数模板外，也支持类模板(class template)。也就是说，可以将某些类定义中的数据类型参数化，设计成类模板。使用时，用具体的数据类型替换模板的参数，从而得到具体的类，这种具体类称为类模板实例(class template specialization)，也称为模板类。模板类与一般类一样，可以实例化为对象。类模板、模板类和对象之间的关系如图 18-2 所示。

在第 11 章用类 Stack 实现了一个抽象数据类型——栈，从程序代码可以看到，该类能够方便地处理元素是整型的数据。但是，如果需要一个元素是 double 类型，或者是自定义

的数据类型(如某种结构),该类就不能满足需要了。因此,必须为每种数据类型设计一个相应的类来实现元素是该类型的栈类。可以想象,这样设计出来的类是非常相似的,如都具有相同的属性、操作功能和约束条件,甚至连类的成员函数的逻辑功能也非常相似,差别仅在于元素的类型不同而已。下面,将栈这种非常通用的抽象数据类型设计成类模板,通过实例化得到处理不同类型的数据的栈。

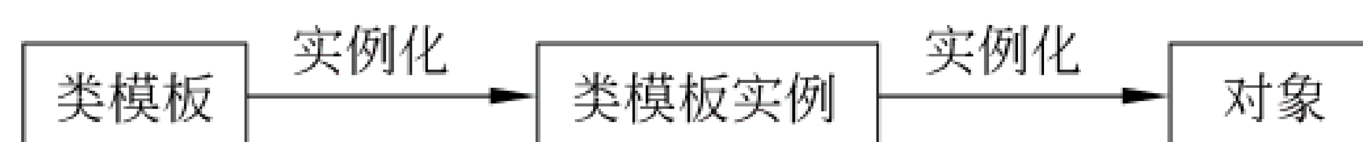


图 18-2　类模板、模板类和对象之间的关系

18.3.1　类模板的定义

在 C++ 程序中,类模板的定义与函数模板的定义类似,也是以关键字 template 开头,随后是用尖括号<>括起来的模板的形式参数列表,每个形式参数之前都有一个关键字 class 或 typename 标识,形式参数名为有效的标识符。

模板的形式参数名可用来说明类的数据成员的类型,也可用来说明成员函数的参数、返回值和函数内的局部变量的类型。形式参数名的作用域局限于该模板的范围内。

模板的参数定义之后是类模板的定义,类模板的定义与一般类的定义相似,只是在定义类模板时,可以用模板的形式参数来说明数据类型。

下面以栈这种抽象数据类型为例,用类模板来处理同元素类型的栈。

【例 18-2】　类模板的定义。

```
//tstack.h:  类模板的定义
#ifndef TSTACK_H
#define TSTACK_H

#define DEFAULT_ELEM_NUM 8

template< class ElementType>
class Stack {
public:
    Stack(int= DEFAULT_ELEM_NUM);                  //缺省栈元素的数目为 8
    ~Stack() { delete [] data;};
    int pop(ElementType &num);
    int push(ElementType num);
private:
     ElementType * data;                           //栈数据存储
     int memNum;                                   //栈元素个数
     int size;                                     //栈大小
};

template< class ElementType>
Stack< ElementType> ::Stack(int s)
{
     size= s> 0 ?s:DEFAULT_ELEM_NUM;
     data= new ElementType[s];
```

```
    memNum=0;
}
//弹栈函数,不成功返回 0,成功则返回 1,栈元素由参数返回
template<class ElementType>
int Stack<ElementType>::pop(ElementType &num)
{
    if (memNum==0)
        return 0;
    num=data[--memNum];
    return 1;
}
//压栈函数,不成功返回 0,成功则返回 1
template<class ElementType>
int Stack<ElementType>::push(ElementType mem)
{
    if (memNum==size)
        return 0;
    data[memNum++]=mem;
    return 1;
}
#endif
```

上述类模板将栈元素的数据类型抽象成模板的参数 ElementType,类模板的定义中用 ElementType 来声明数据成员 data、成员函数 pop 和 push 的形式参数的类型。实际使用类模板时,模板的形式参数将被具体的数据类型所替换,从而将类模板实例化成具体的类,可以用这些实例化的类来建立相应的对象。

需要特别说明的是,除在类模板的声明中定义的成员函数以外,所有在类模板外定义的成员函数都要以下面的形式开头:

```
template <class ElementType>
```

而且成员函数定义时,二元作用域运算符::之前的类名要用 Stack<ElementType>,这样就能将成员函数的定义与类模板以及模板参数联系起来。

18.3.2　使用类模板

类模板必须实例化才能够使用。将类模板实例化的过程,就是用具体的数据类型替换模板的参数,而得到具体的类。用具体的数据类型实例化类模板的形式如下:

```
类模板名 <数据类型列表>
```

<数据类型列表>由多个数据类型构成,它们之间用逗号分隔,分别对应模板的参数。编译器对模板实例化时,将类模板中的所有类型参数替换成相应具体数据类型,将该类模板实例化成一个具体的类,并进行编译。

例如,对于上面定义的类模板 Stack,可以按下面方式对其进行实例化:

```
Stack <double>
```

即用 double 替换类型参数 ElementType,将该类模板实例化成具体的类,相当于定义了一个元素为 double 的栈类。实例化的类在使用上与一般的类完全一样,也可用来建立相应的对象。例如,可用刚才实例化的类建立一个名字为 doubleStack 的对象:

```
Stack <double>  doubleStack(6);
```

doubleStack 对象在创建时,用 6 作为参数调用构造函数,这与建立一般类的对象一样。当然,也可以将 Stack 实例化成处理 int 类型元素的栈,并创建相应的对象:

```
Stack <int>  intStack;
```

编译器处理该语句时,首先用 int 实例化 Stack 类模板,再用该实例化的类声明一个名为 intStack 的对象,并为该对象以默认参数调用构造函数。

doubleStack 和 intStack 这类对象的使用与一般类的对象的使用完全一样。

下面编写一个使用上一节定义的 Stack 类模板的程序。

【例 18-3】 用类模板实现元素为不同类型(double 和 int)的栈。

```
//ex18_3:  用类模板实现元素为不同类型(double 和 int)的栈
#include <iostream.h>
#include "tstack.h"

int main()
{
    //用 double 实例化 Stack 模板类,并建立实例化类的对象
    Stack< double>  doubleStack(6);

    //使用 doubleStack
    //向 doubleStack 中压入 double 数据
    double f= 3.14;
    cout<< "Pushing elements onto doubleStack:\n";

    while (doubleStack.push(f))
    {
        cout<< f<< ' ';
        f += f;
    }
    cout<< "\nStack is full,can not push "<< f<< " onto the doubleStack.";

    //从 doubleStack 中弹出 double 数据
    cout<< "\n\nPopping elements from doubleStack:\n";
    while (doubleStack.pop(f))
        cout<< f<< ' ';
    cout<< "\nStack is empty,can not pop.\n";

    //用 int 实例化 Stack 模板类,并建立实例化类的对象
    Stack< int>  intStack;

    //使用 intStack
    //向 intStack 中压入 int 数据
```

```
    int i=1;
    cout<<"\nPushing elements onto intStack:\n";
    while (intStack.push(i))
    {
        cout<<i<<' ';
        ++i;
    }
    cout<<"\nStack is full,can not push "<<i<<" onto the intStack.";
    //从 doubleStack 中弹出 double 数据
    cout<<"\n\nPopping elements from intStack:\n";
    while (intStack.pop(i))
      cout<<i<<' ';
    cout<<"\nStack is empty,can not pop.\n";
    return 0;
}
```

程序执行结果：

```
Pushing elements onto doubleStack:
3.14  6.28  12.56  25.12  50.24  100.48
Stack is full,can not push 200.96 onto the doubleStack.
Popping elements from doubleStack:
100.48  50.24  25.12  12.56  6.28  3.14
Stack is empty,can not pop.
Pushing elements onto intStack:
1  2  3  4  5  6  7  8
Stack is full,can not push 9 onto the intStack.
Popping elements from intStack:
8  7  6  5  4  3  2  1
Stack is empty,can not pop.
```

从上述程序可以看出，通过定义 Stack 类模板，可以灵活地将其实例化成不同的类，这些实例化的类在使用上与一般的类完全一样，可以创建自己的实例，而这些对象也与一般类的对象完全一样。

通过上面的学习可以看出，模板提供了更高层次的抽象机制，提高了软件重用的程度。利用模板机制，可以建立起非常有效的类库，为程序设计提供更强大的支持。事实上，标准C++语言本身就提供了一个标准模板库(standard template library，STL)，包括处理 list、vector、set 和 map 等数据结构的类模板、算法(如通用的遍历、排序以及合并算法)和其他组件，为程序设计者提供了丰富的功能。

习　题　18

18.1　分析函数模板和函数重载之间的关系。

18.2　编写一个函数模板，从一维数组中查找值为给定值的元素，若查找成功则返回真，否则返回假。编

写一个使用该模板的程序,用不同的数据类型实例化该模板。

18.3 编写一个函数模板,实现将任意数组的元素倒置,即将数组的首尾逆转。并编写一个使用该模板的程序,实现对不同类型的数组的倒置。

18.4 编写一个类模板,该类模板可以用各种类型实例化成元素是该类型的数组类,要求数组类提供下列功能:

(1) 根据下标取数组元素的值。

(2) 根据下标设置数组元素的值。

(3) 根据下标在指定位置插入的数组元素。

(4) 根据下标删除指定位置的数组元素。

(5) 在数组尾部增加元素。

为了说明类模板的功能,请编写一个程序使用该类模板实现对不同类型的数组的各种操作。

附录 A　C++ 运算符的优先级和结合性

表 A-1 按照运算符的优先级从高到低排列。

表 A-1　C++ 运算符的优先级和结合性

优　先　级	运　算　符	描　　述	结　合　性
17	::	二元作用域	从左向右
	::	一元作用域	从右向左
16	()	括号	从左向右
	[]	数组下标	
	.	通过对象名选择成员	
	－＞	通过指针选择成员	
	＋＋	一元后自增	
	－－	一元后自减	
	typeid	运行时类型信息	
	dynamic_cast＜type＞	运行时类型检查的强制类型转换	
	static_cast＜type＞	编译时类型检查的强制类型转换	
	reinterpret_cast＜type＞	非标准转换的强制类型转换	
	const_cast＜type＞	对常量性进行强制类型转换	
15	＋＋	一元前自增	从右向左
	－－	一元前自减	
	＋	一元正	
	－	一元负	
	!	一元逻辑非	
	～	一元按位取反	
	(type)	一元强制类型转换	
	sizeof	确定字节数	

续表

优 先 级	运 算 符	描 述	结 合 性
15	&	取地址	从右向左
	*	间接引用	
	new	动态内存分配	
	new []	动态数组分配	
	delete	动态内存释放	
	delete []	动态数组释放	
14	.*	通过对象取指向成员的指针	从左向右
	－＞*	通过指针取指向成员的指针	
13	*	乘	从左向右
	/	除	
	%	求余	
12	＋	加	从左向右
	－	减	
11	＜＜	向左移位	从左向右
	＞＞	向右移位	
10	＜	小于	从左向右
	＜＝	小于或等于	
	＞	大于	
	＞＝	大于或等于	
9	＝＝	等于	从左向右
	!＝	不等于	
8	&	按位与	从左向右
7	^	按位异或	从左向右
6	|	按位或	从左向右
5	&&	逻辑与	从左向右
4	||	逻辑或	从左向右
3	?:	三元条件运算	从右向左
2	＝	赋值	从右向左
	＋＝	加法赋值	
	－＝	减法赋值	

续表

优　先　级	运　算　符	描　　述	结　合　性
2	*=	乘法赋值	从右向左
	/=	除法赋值	
	%=	求余赋值	
	&=	按位与赋值	
	^=	按位异或赋值	
	|=	按位或赋值	
	<<=	向左移位赋值	
	>>=	向右移位赋值	
1	,	逗号	从左向右

附录 B ASCII 字符集

ASCII(American Standard Code for Information Interchange)字符集和 ASCII 码被国际标准化组织(International Organization for Standardization,ISO)批准为国际标准。基本的 ASCII 字符集共有 128 个字符,其中有 96 个可打印字符,包括常用的字母、数字、标点符号和空白符等,另外还有 33 个控制字符。标准 ASCII 码使用 7 个二进位对字符进行编码,对应的 ISO 标准为 ISO646 标准。表 B-1 给出了基本 ASCII 字符及其编码。

表 B-1 ASCII 字符及其编码

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

表 B-1 左边的数字代表字符编码的高 3 位,上边的数字代表编码的低 4 位。表中 34 个特殊字符(包括 SP 和 33 个控制字符)的意义如表 B-2 所示。

表 B-2 特殊字符说明

二 进 制	十 进 制	十六进制	缩 写	意 义
0000 0000	0	00	NUL	Null Character
0000 0001	1	01	SOH	Start of Header
0000 0010	2	02	STX	Start of Text
0000 0011	3	03	ETX	End of Text
0000 0100	4	04	EOT	End of Transmission

续表

二　进　制	十　进　制	十六进制	缩　　写	意　　义
0000 0101	5	05	ENQ	Enquiry
0000 0110	6	06	ACK	Acknowledgment
0000 0111	7	07	BEL	Bell
0000 1000	8	08	BS	Backspace
0000 1001	9	09	HT	Horizontal Tab
0000 1010	10	0A	LF	Line feed
0000 1011	11	0B	VT	Vertical Tab
0000 1100	12	0C	FF	Form Feed
0000 1101	13	0D	CR	Carriage return
0000 1110	14	0E	SO	Shift Out
0000 1111	15	0F	SI	Shift In
0001 0000	16	10	DLE	Data Link Escape
0001 0001	17	11	DC1	XON Device Control 1
0001 0010	18	12	DC2	Device Control 2
0001 0011	19	13	DC3	XOFF Device Control 3
0001 0100	20	14	DC4	Device Control 4
0001 0101	21	15	NAK	Negative Acknowledgement
0001 0110	22	16	SYN	Synchronous Idle
0001 0111	23	17	ETB	End of Trans. Block
0001 1000	24	18	CAN	Cancel
0001 1001	25	19	EM	End of Medium
0001 1010	26	1A	SUB	Substitute
0001 1011	27	1B	ESC	Escape
0001 1100	28	1C	FS	File Separator
0001 1101	29	1D	GS	Group Separator
0001 1110	30	1E	RS	Record Separator
0001 1111	31	1F	US	Unit Separator
0010 0000	32	20	SP	Space
0111 1111	127	7F	DEL	Delete

参 考 文 献

［1］ Programming Languages—C++ (Second Edition). Reference number：ISO/IEC 14882:2003(E)，2003.

［2］ Bruce Eckel. C++编程思想. 第 2 版，影印版. 北京：机械工业出版社，2002.

［3］ Harvey M Deitel，Paul James Deitel. C++大学教程. 第 2 版. 邱仲潘等译. 北京：电子工业出版社，2001.

［4］ Bjarne Stroustrup. C++程序设计语言. 特别版，影印版. 北京：高等教育出版社，2001.

［5］ 中国大百科全书——计算机卷. 北京：中国大百科全书出版社，2000.

［6］ 谭浩强. C 程序设计. 第 2 版. 北京：清华大学出版社，2000.

［7］ Brian W Kernighan. C 程序设计语言. 第 2 版. 徐宝文译. 北京：机械工业出版社，2001.

［8］ Nicolai M Josuttis. The C++ Standard Library—A Tutorial and Reference. Addison-Wesley U. S. A. ，1999.



普通高等教育"十一五"国家级规划教材
21世纪大学本科计算机专业系列教材

近期出版书目

- 计算概论(第2版)
- 计算概论——程序设计阅读题解
- 计算机导论(第3版)
- 计算机导论教学指导与习题解答
- 计算机伦理学
- 程序设计导引及在线实践
- 程序设计基础(第2版)
- 程序设计基础习题解析与实验指导
- 程序设计基础(C语言)(第2版)
- 程序设计基础(C语言)习题集与实验指导(第2版)
- 离散数学(第3版)
- 离散数学习题解答与学习指导(第3版)
- 数据结构(STL框架)
- 算法设计与分析
- 算法设计与分析习题解答与学习指导
- 算法设计与分析(第3版)
- 算法设计与分析习题解答(第3版)
- C++程序设计(第3版)
- Java程序设计
- 面向对象程序设计(第3版)
- 形式语言与自动机理论(第3版)
- 形式语言与自动机理论教学参考书(第3版)
- 数字电子技术基础
- 数字逻辑
- 计算机组成原理(第3版)
- 计算机组成原理教师用书(第3版)
- 计算机组成原理学习指导与习题解析(第3版)
- 微机原理与接口技术
- 微型计算机系统与接口(第2版)
- 计算机组成与系统结构
- 计算机组成与体系结构习题解答与教学指导
- 计算机组成与体系结构(第2版)
- 计算机系统结构教程
- 计算机系统结构学习指导与题解
- 计算机操作系统(第2版)
- 计算机操作系统学习指导与习题解答
- 编译原理
- 软件工程(第2版)
- 计算机图形学
- 计算机网络(第3版)
- 计算机网络教师用书(第3版)
- 计算机网络实验指导书(第3版)
- 计算机网络习题解析与同步练习
- 计算机网络软件编程指导书
- 人工智能
- 多媒体技术原理及应用(第2版)
- 计算机网络工程(第2版)
- 计算机网络工程实验教程
- 信息安全原理及应用